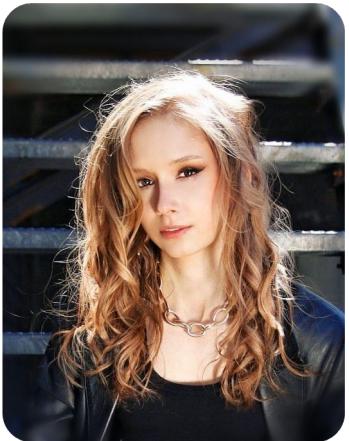


Intro to transformers



Maria Tikhonova

Senior NLP Data Scientist, SberDevices

Invited Lecturer, HSE

HI!

My name is Maria Tikhonova

Graduated from MSU & YSDA

Work as a Senior NLP Data Scientist in

Teach in



Outline

- Motivation for Transformer
- Attention (from RNNs to Attention is all you need)
- The original Transformer architecture
- Do we need a transformer for our problem?

Why transformers?

- Natural language processing ≈ understanding and generating texts
- One of the most important NLP problems: machine translation
 - “Je m'appelle Mari.” ↗ “My name is Maria.”
- What one needs for text translation?
 1. Understand individual words Subword vocabulary + embeddings
 2. Understand interaction between words (syntax) Encoder self-attention
 3. Translate the words (given the context) Cross-attention
 4. Compose a meaningful and fluent text Encoder self-attention
- The transformer architecture offers a **unified, powerful, and reusable way of text understanding and generation**

Short history of machine
translation.

Motivation for transformers

History of machine translation

- 1950s: Russian→English MT (Cold War)
 - *Rule-based systems using bilingual dictionaries*

History of machine translation

- 1950s: Russian→English MT (Cold War)
 - *Rule-based systems using bilingual dictionaries*
- 1990s-2010s: **Statistical** MT (SMT)
 - *Mainly **Phrase-Based** MT (PBMT)*
 - *Learns from (large) **sentence-aligned** bilingual corpora*
 - *Consists of separate very complex components, learnt separately*
 - *Developed by large groups for decades, doesn't generalize to new language pairs*

History of machine translation

- 1950s: Russian→English MT (Cold War)
 - *Rule-based systems using bilingual dictionaries*
- 1990s-2010s: **Statistical** MT (SMT)
 - *Mainly **Phrase-Based** MT (PBMT)*
 - *Learns from (large) **sentence-aligned** bilingual corpora*
 - *Consists of separate very complex components, learnt separately*
 - *Developed by large groups for decades, doesn't generalize to new language pairs*
- Since 2014: **Neural** MT (NMT)
 - *Single NN learnt **end-to-end***
 - *Learns from (large) **sentence-aligned** bilingual corpora*
 - *A few (good) student / month to implement*
 - *better quality of translation than SMT systems developed for decades*
- Google Translate: NMT, Yandex Translate: NMT+PBMT

History of machine translation: IBM models

Original French sentence

English translation

$$\Pr(\mathbf{e}|\mathbf{f}) = \frac{\Pr(\mathbf{e}) \Pr(\mathbf{f}|\mathbf{e})}{\Pr(\mathbf{f})}$$

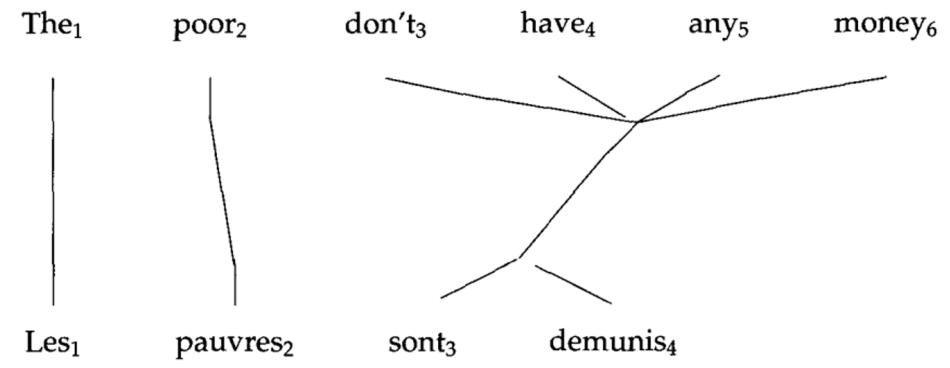
English N-gram language model

Translation model

This probability can be discarded

$$\hat{\mathbf{e}} = \operatorname{argmax}_{\mathbf{e}} \Pr(\mathbf{e}) \Pr(\mathbf{f}|\mathbf{e})$$

Key model component: learned word alignment



$$\Pr(\mathbf{f}|\mathbf{e}) = \sum_{\mathbf{a}} \Pr(\mathbf{f}, \mathbf{a}|\mathbf{e})$$

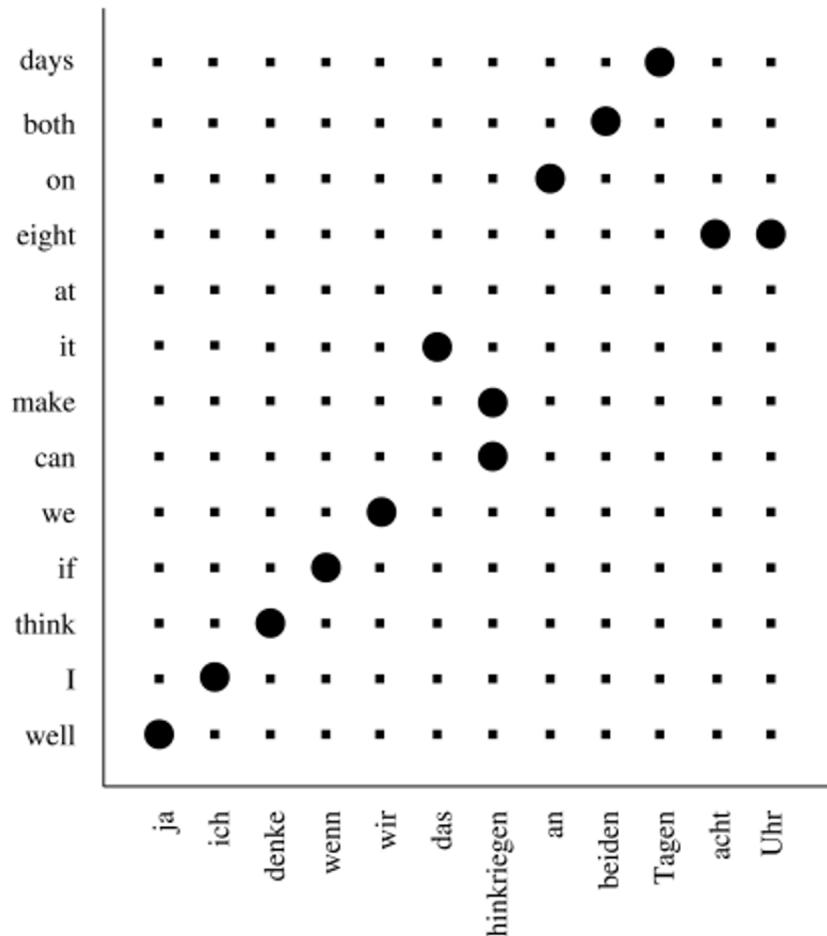
$$\Pr(f, a|e) = \Pr(f|a, e) \Pr(a|e)$$

Phrase-by-phrase translation

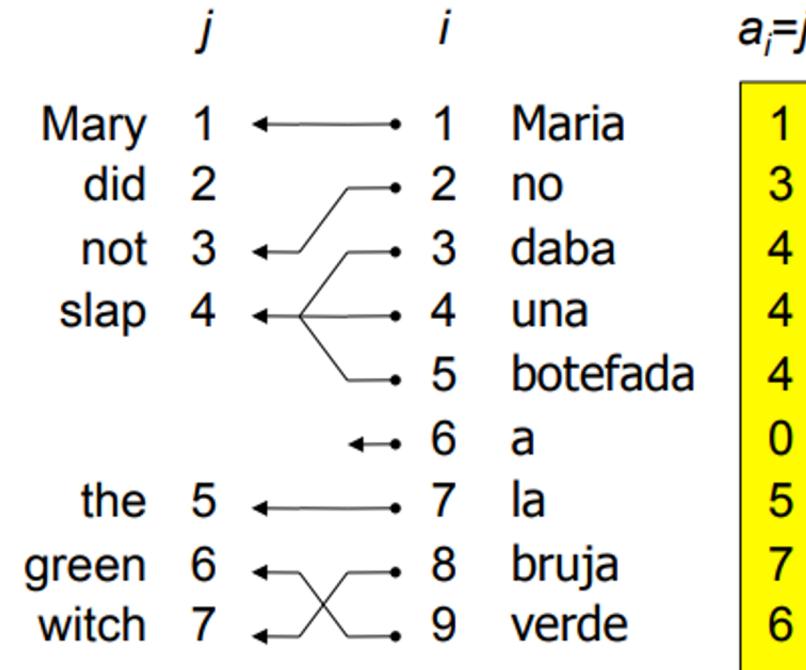
Alignment model

What is an alignment?

A matrix of matches between words

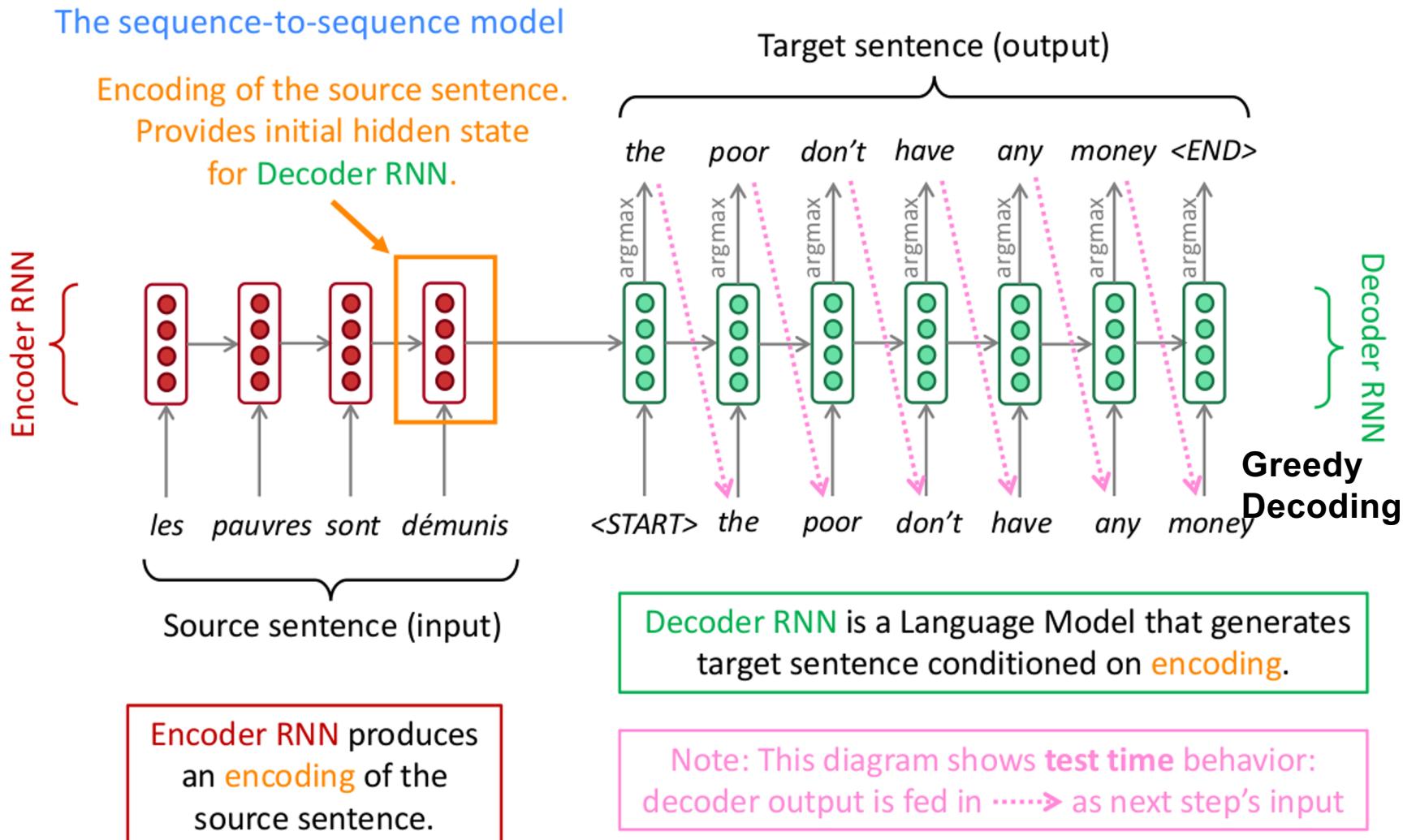


A vector of matched word*



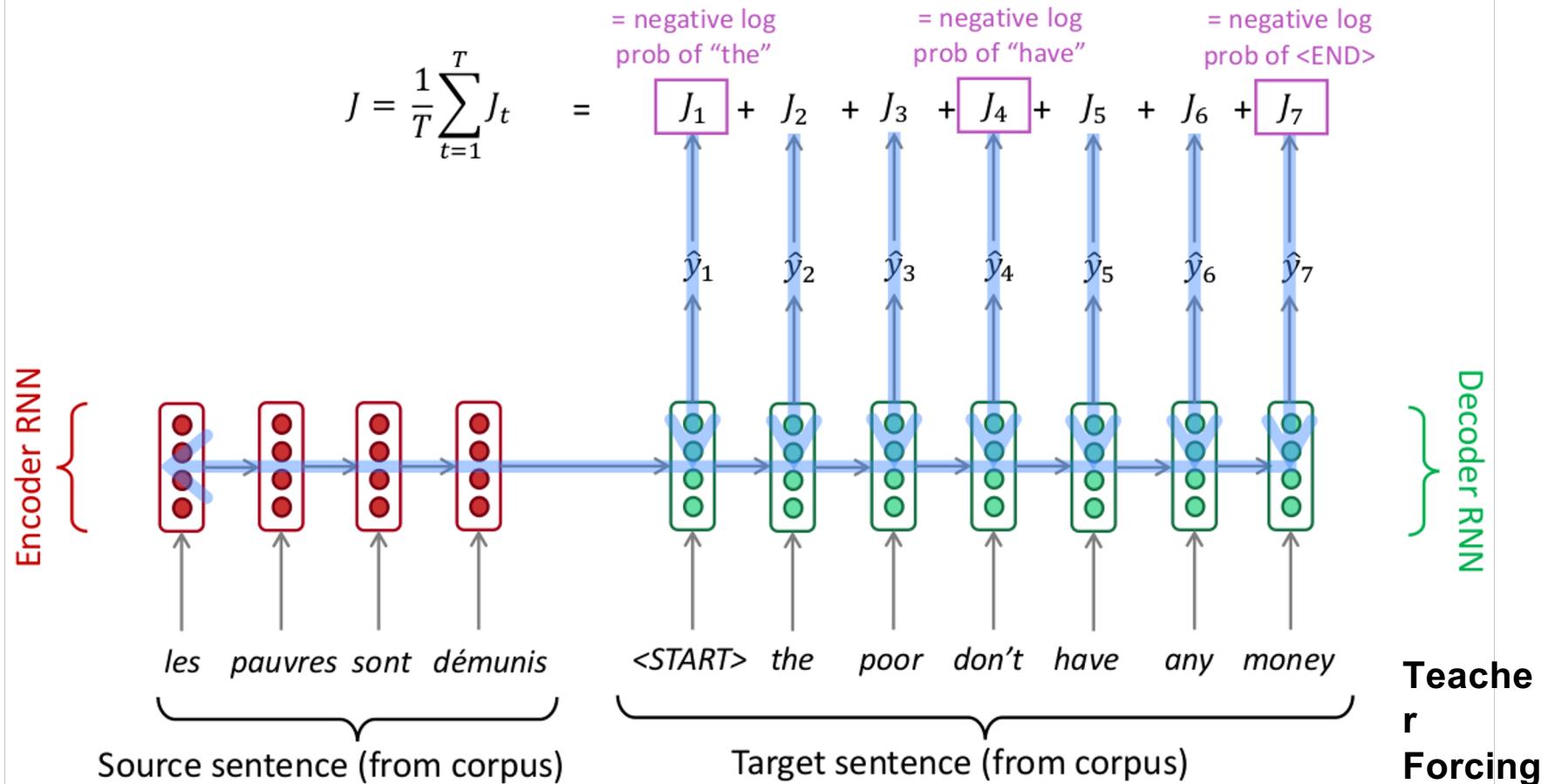
*This approach is simpler, but it cannot describe one-to-many or many-to-many alignments

Neural machine translation: seq2seq

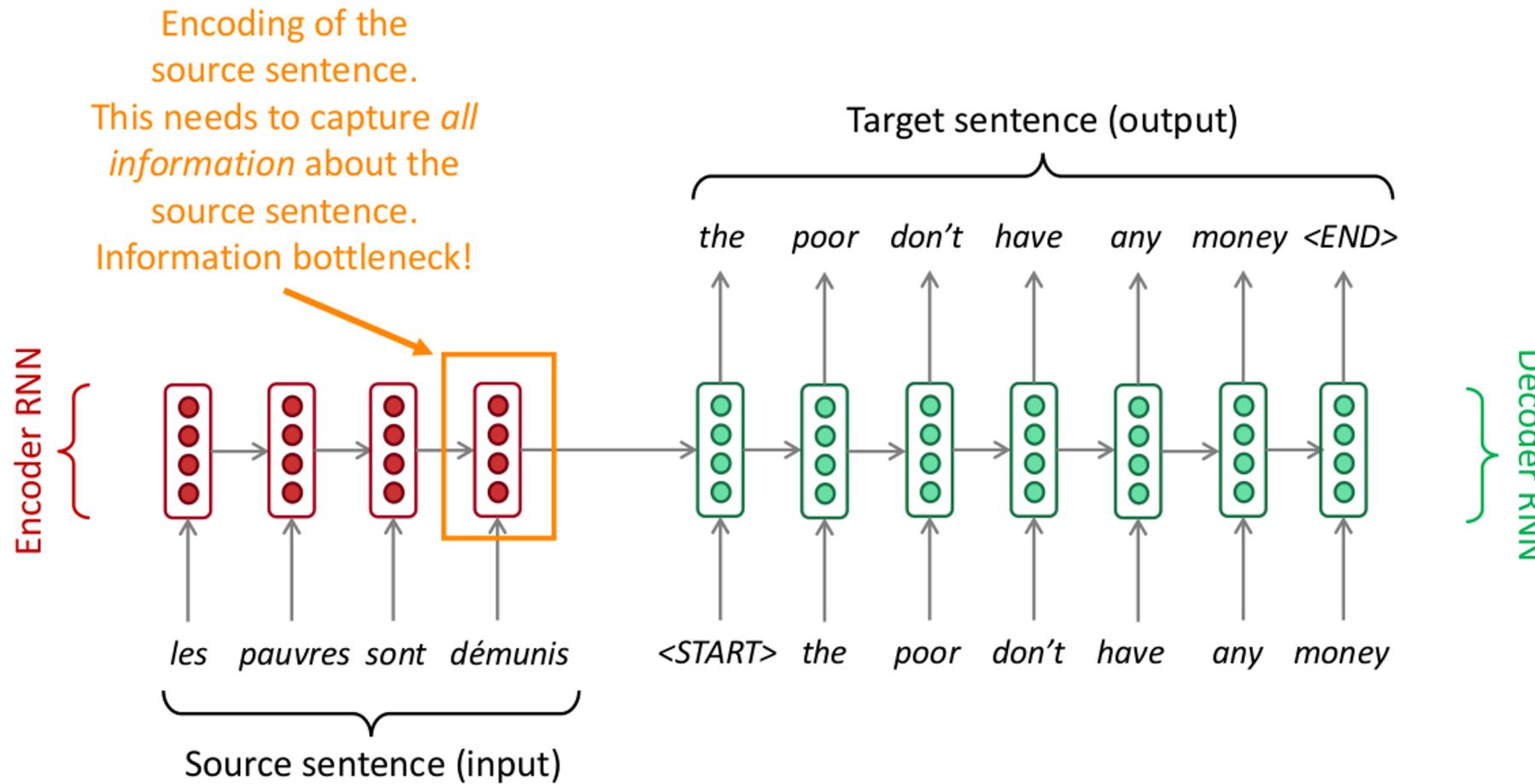


Neural machine translation: seq2seq loss

Training a Neural Machine Translation system



Seq2seq bottleneck problem



Attention

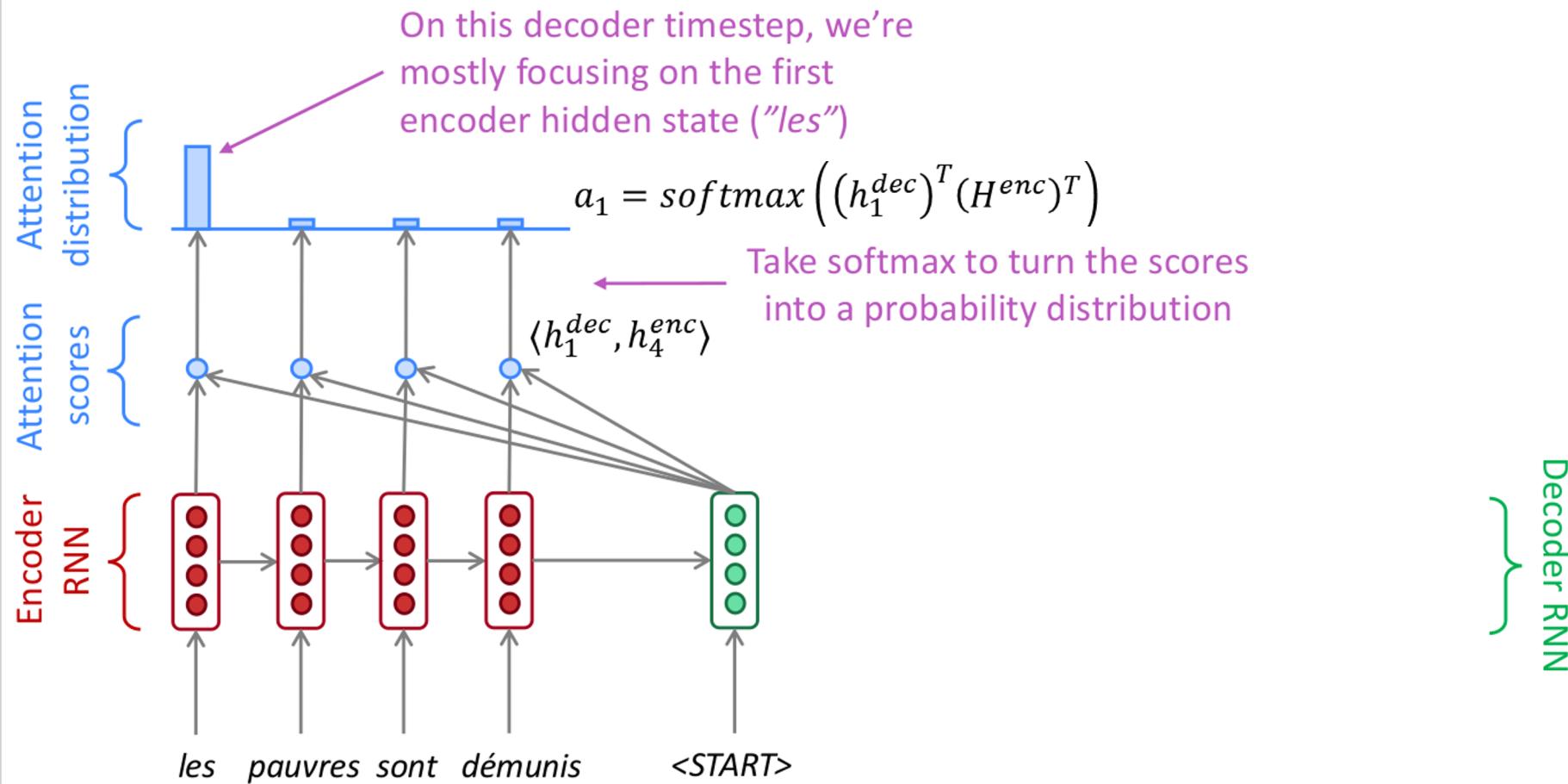
Attention:

At different steps, let a model *focuses* on different parts of the source tokens (more relevant ones).

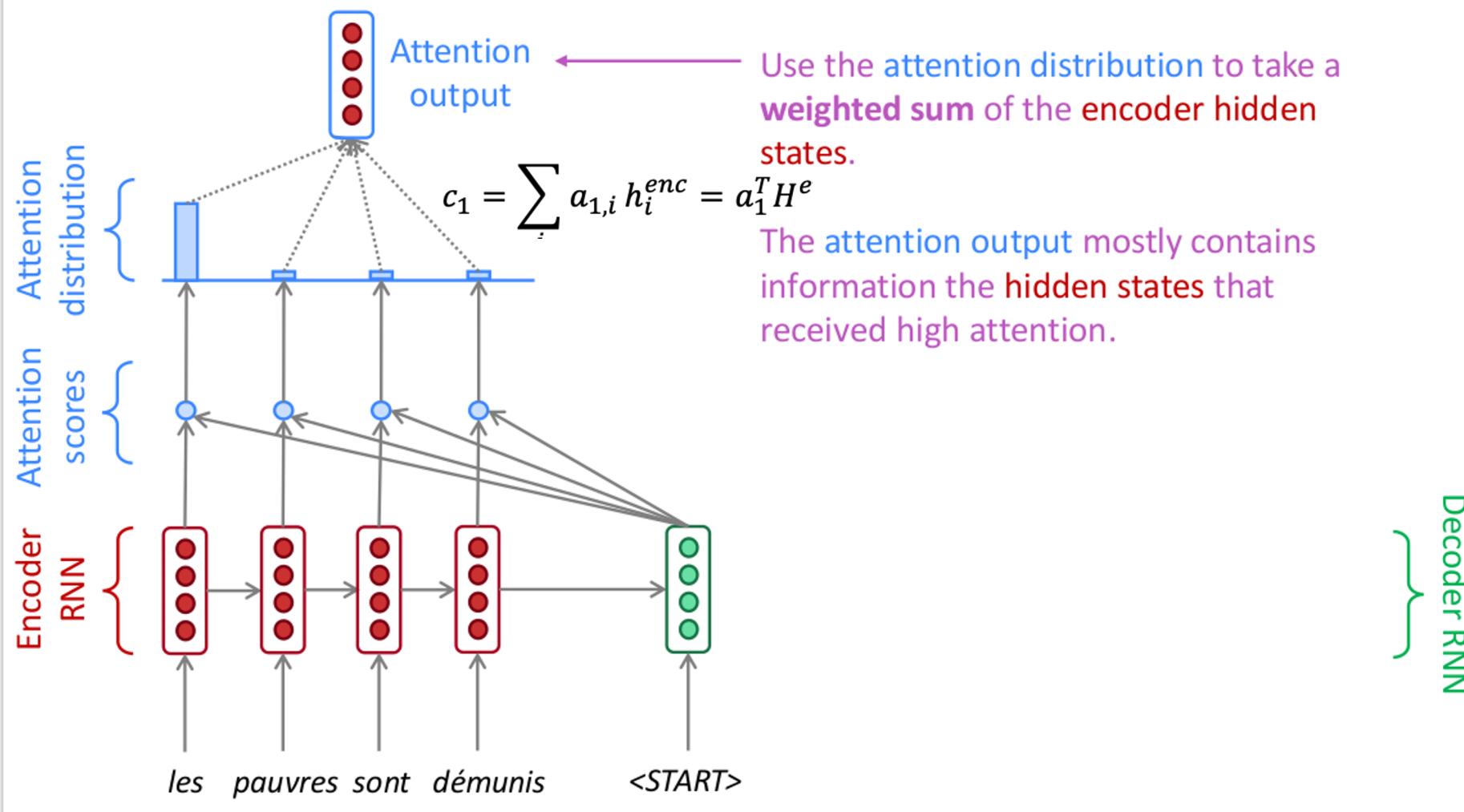
Core idea:

On each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence

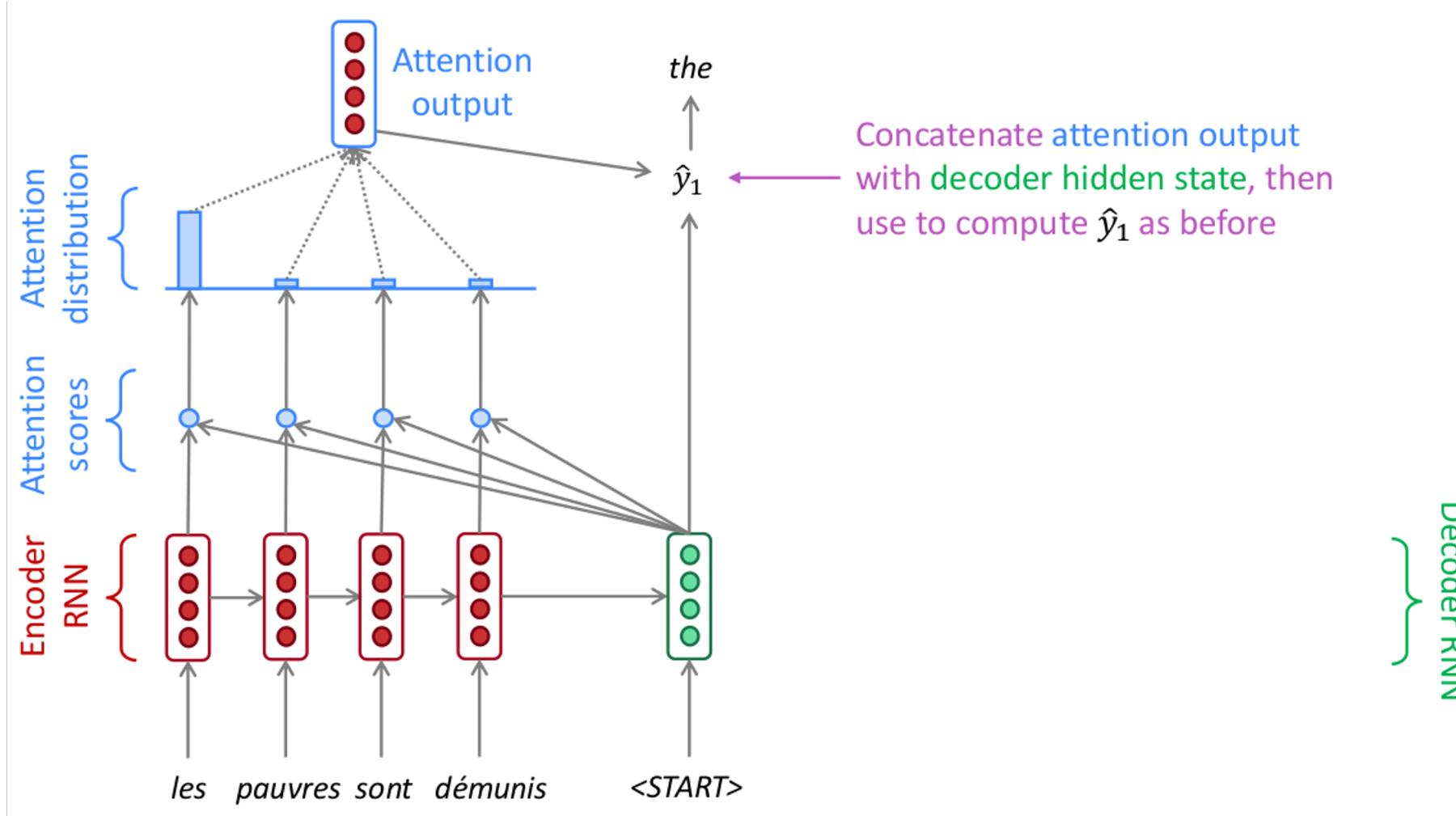
Attention: Example



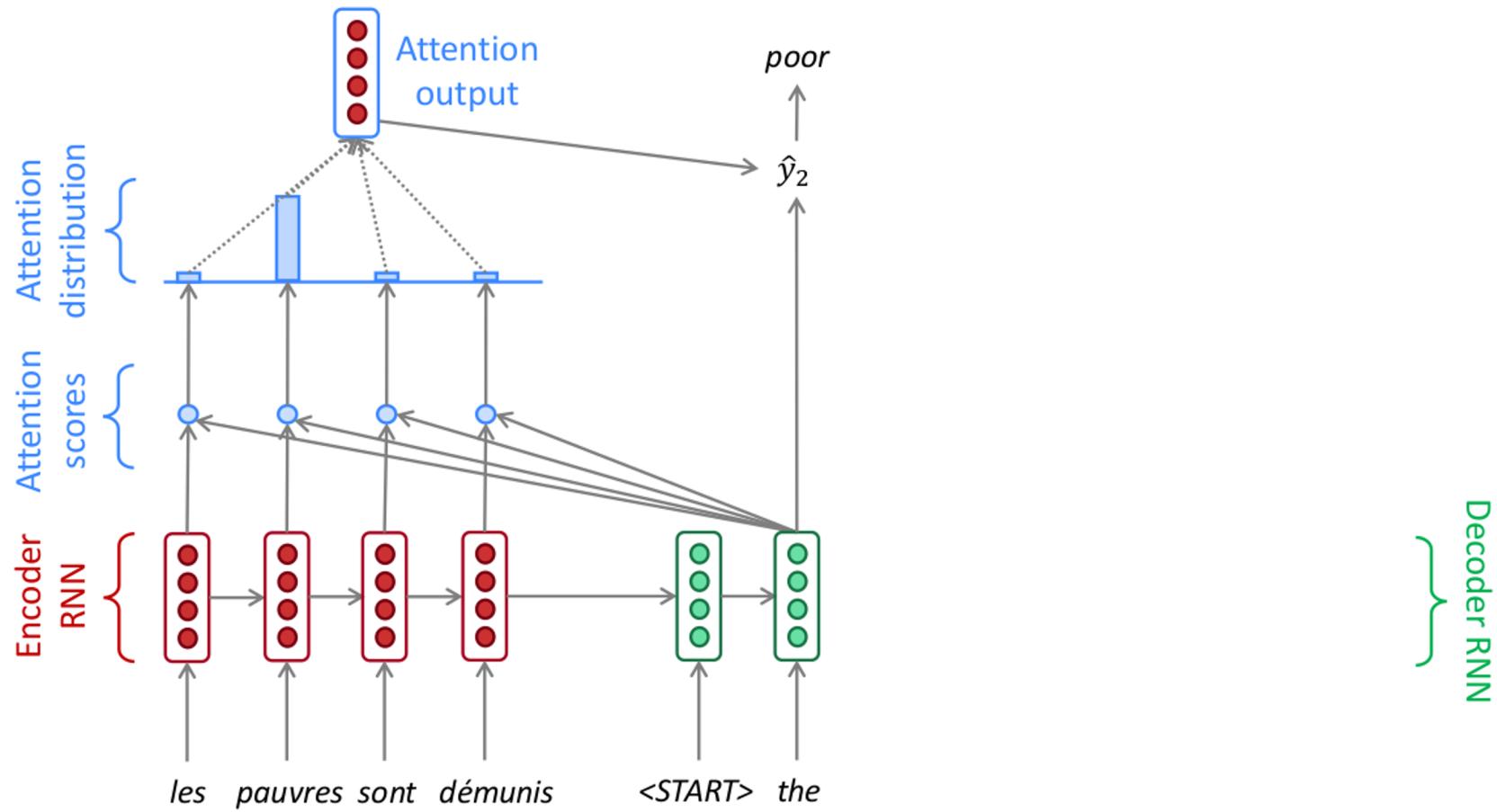
Attention: Example



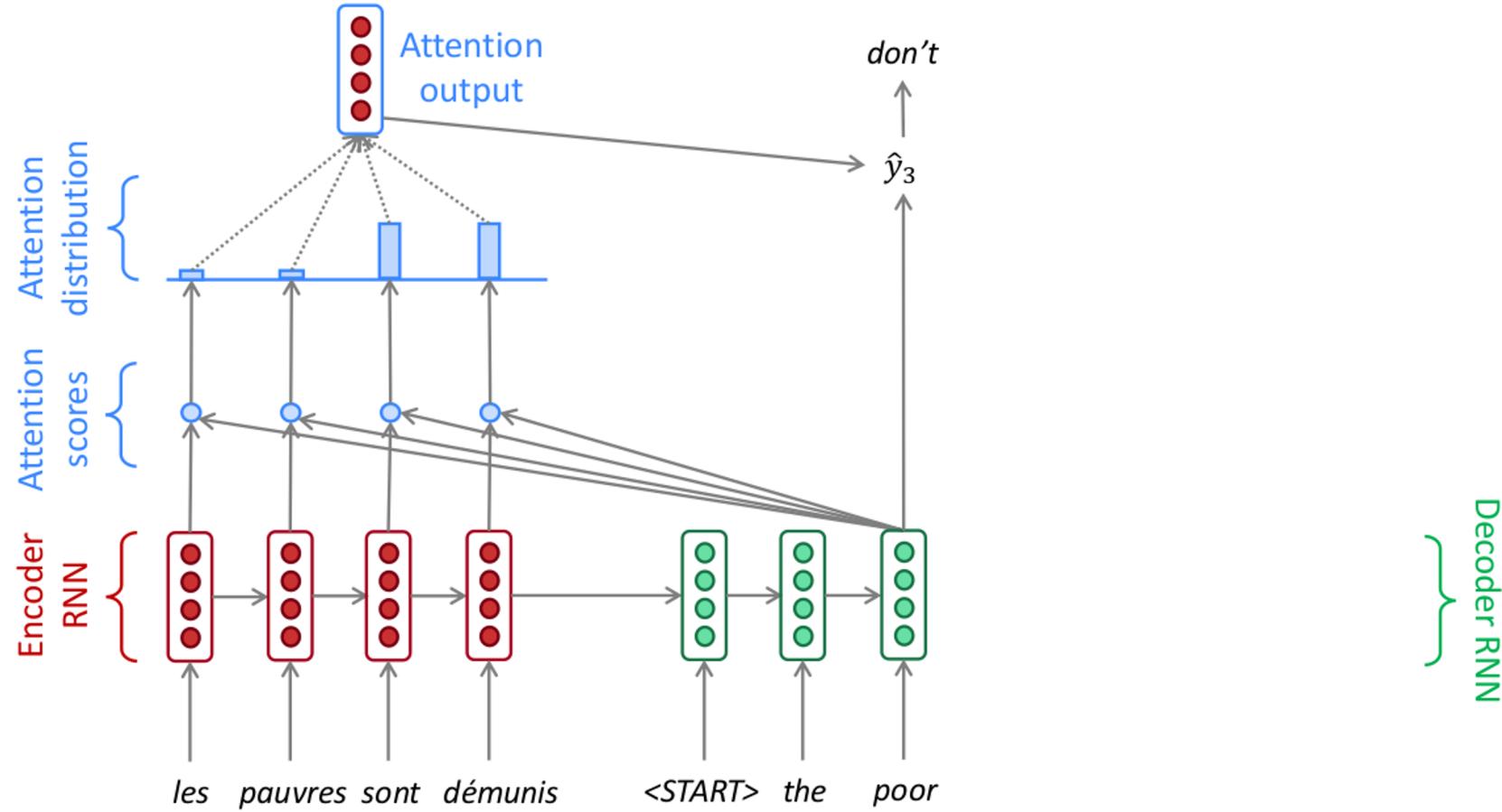
Attention: Example



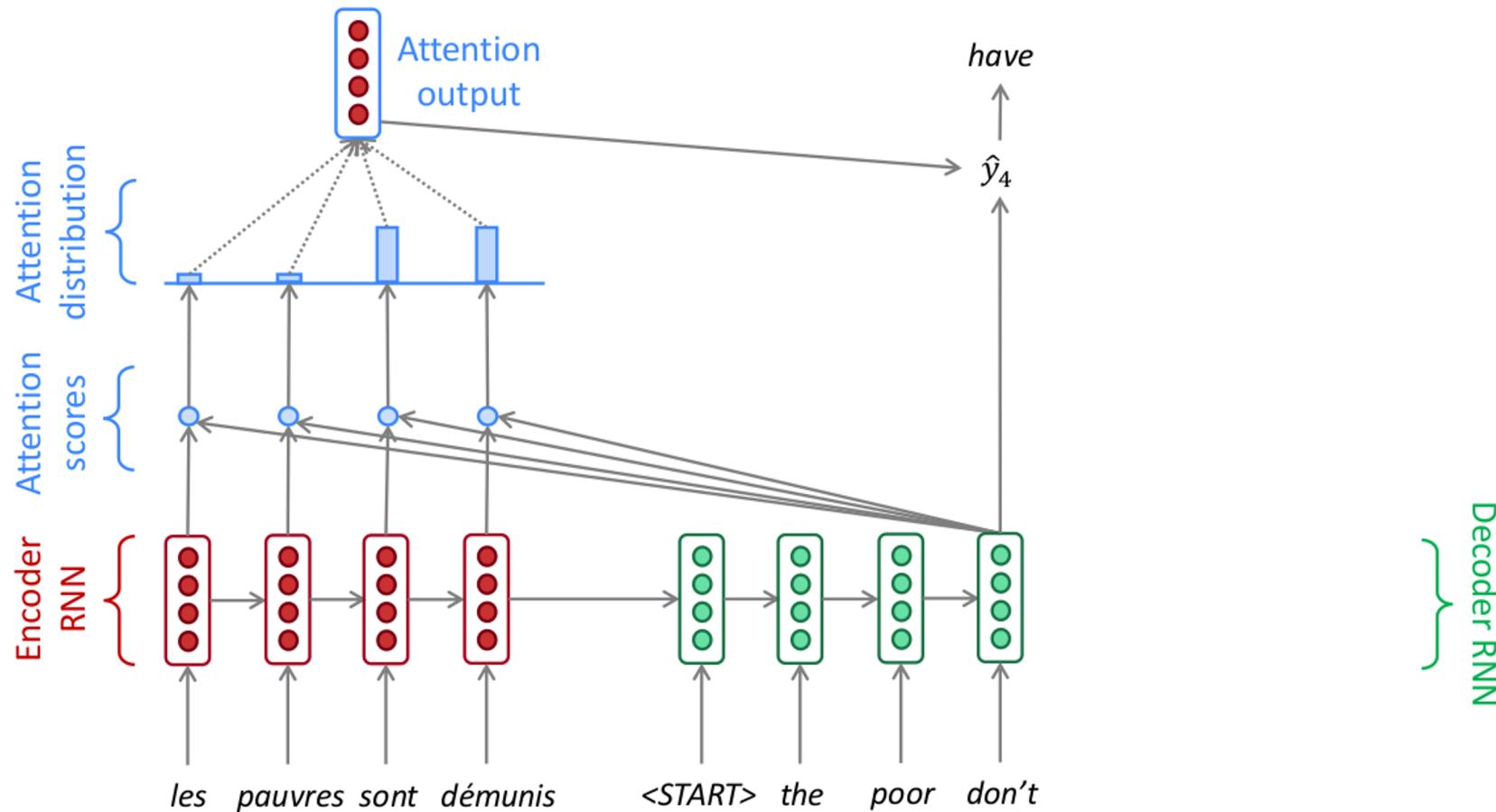
Attention: Example



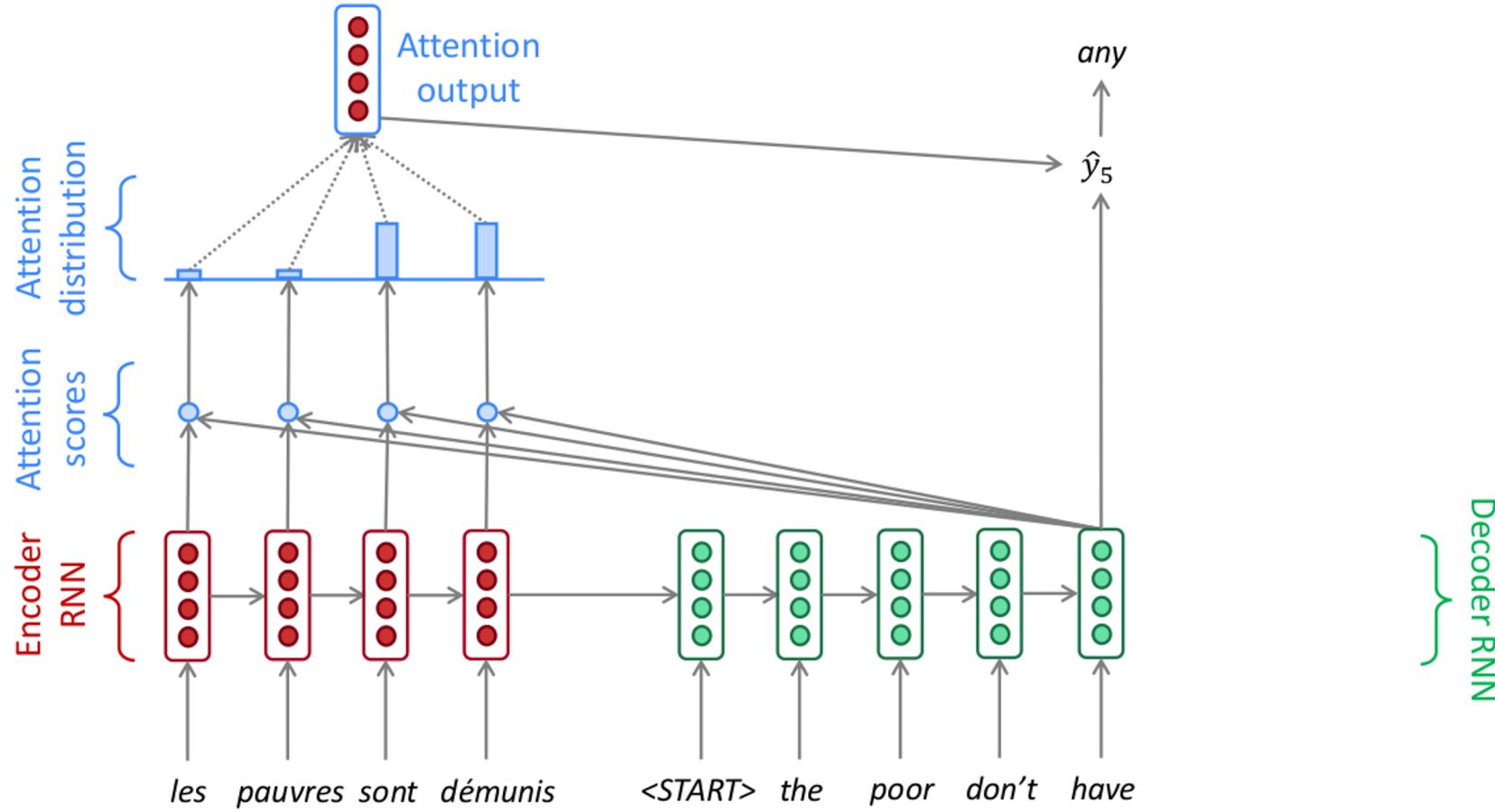
Attention: Example



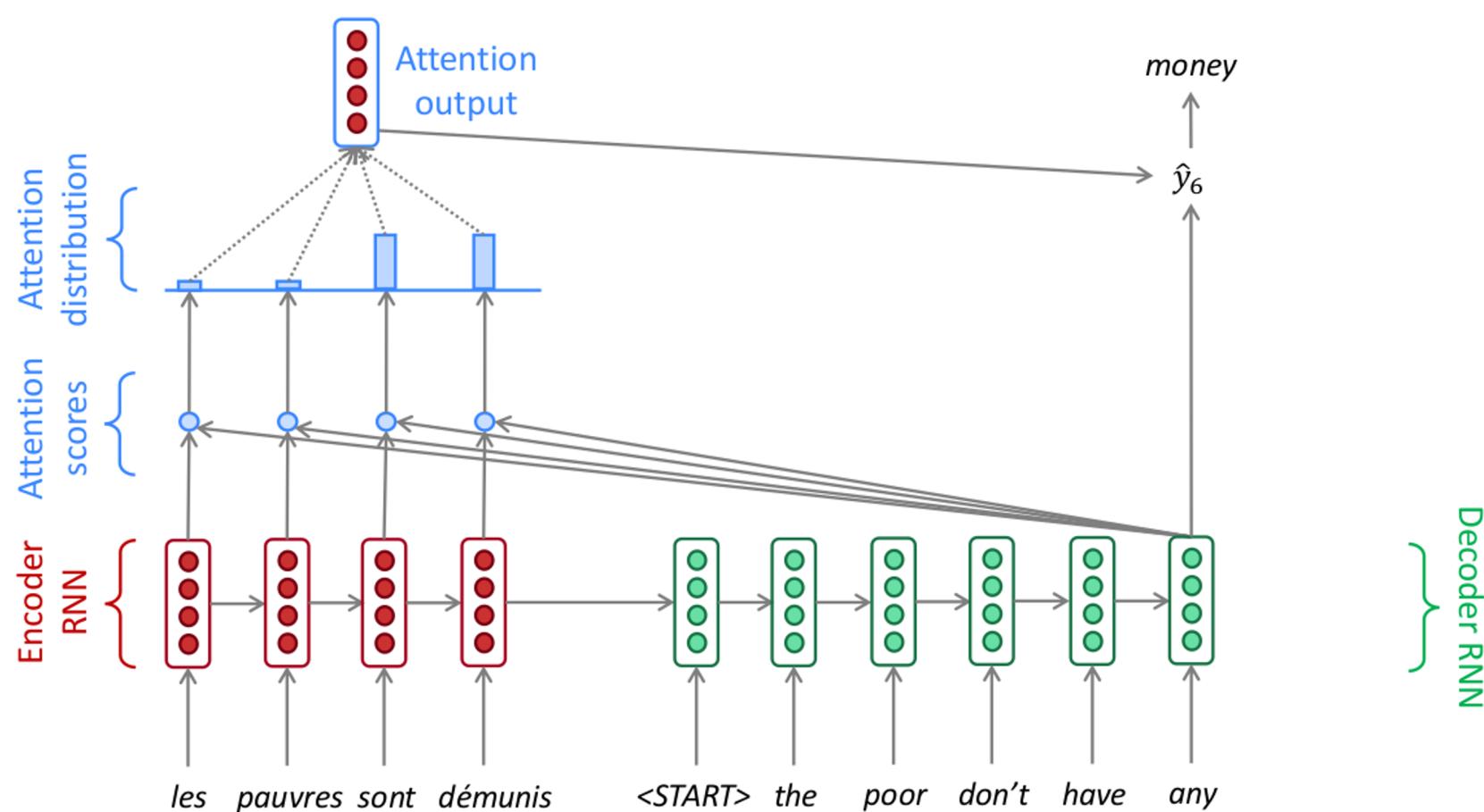
Attention: Example



Attention: Example



Attention: Example



Attention

Results:

- Solves **information bottleneck** problem, acts like an unlimited size memory
- Helps **gradient propagation** from encoder to decoder, especially in long sequences
- Better interpretability

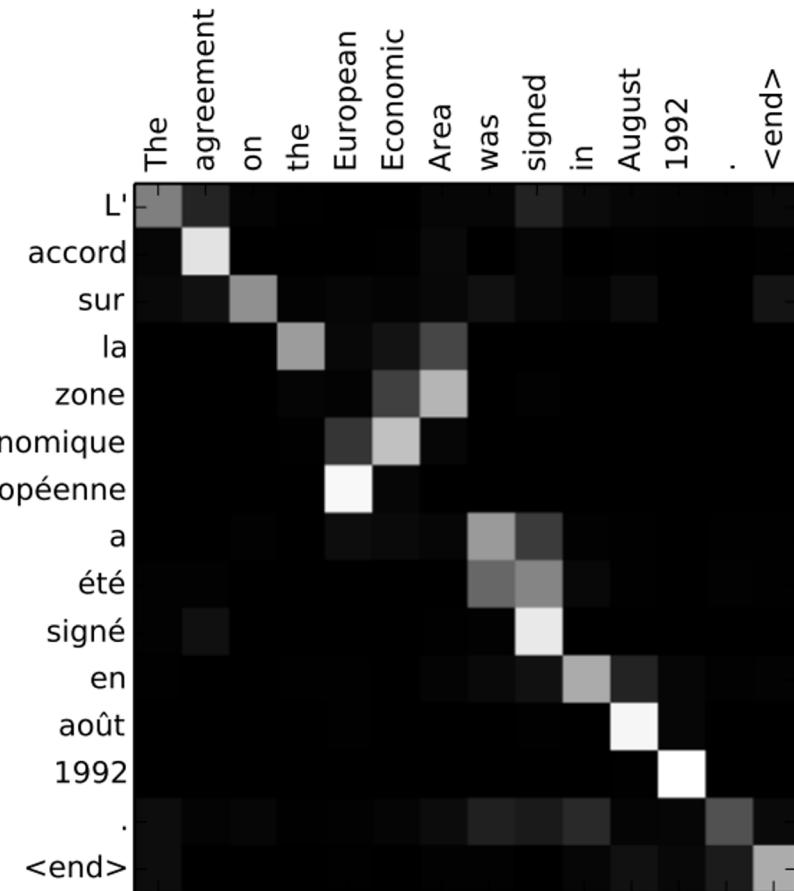
Britz et al., 2017. Massive Exploration of Neural Machine Translation Architectures:

- “we found that the attention-based models exhibited significantly larger gradient updates to decoder states throughout training. This suggests that the attention mechanism acts more like a “weighted skip connection” that optimizes gradient flow than like a “memory” that allows the encoder to access source states, as is commonly stated in the literature”

Attention

Better interpretability

- Networks learn alignment as a byproduct of translation
- We can look at which fragments were translated to which ones



Attention

- Attention is a new basic layer type (along with feed-forward, convolutional and recurrent)
- Works with variable length inputs (texts)
 - *like RNNs, CNNs*
 - *unlike feed-forward*
- Used in SOTA models in lots of tasks (question answering, image captioning, ...)

The Transformer

Get rid of RNNs in MT?

- RNNs are slow, because not parallelizable over timesteps

Attention Is All You Need				Model	BLEU		Training Cost (FLOPs)	
Ashish Vaswani*	Noam Shazeer*	Niki Parmar*	Jakob Uszkoreit*		EN-DE	EN-FR	EN-DE	EN-FR
Google Brain avaswani@google.com	Google Brain noam@google.com	Google Research nikip@google.com	Google Research usz@google.com	ByteNet [18]	23.75			
				Deep-Att + PosUnk [39]		39.2	$1.0 \cdot 10^{20}$	
				GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
				ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
				MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Llion Jones* Google Research llion@google.com	Aidan N. Gomez* [†] University of Toronto aidan@cs.toronto.edu	Lukasz Kaiser* Google Brain lukaszkaiser@google.com		Deep-Att + PosUnk Ensemble [39]		40.4	$8.0 \cdot 10^{20}$	
				GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
				ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
	Illia Polosukhin* [‡] illia.polosukhin@gmail.com			Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
				Transformer (big)	28.4	41.8		$2.3 \cdot 10^{19}$

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

[†]Work performed while at Google Brain.

[‡]Work performed while at Google Research.

The Transformer

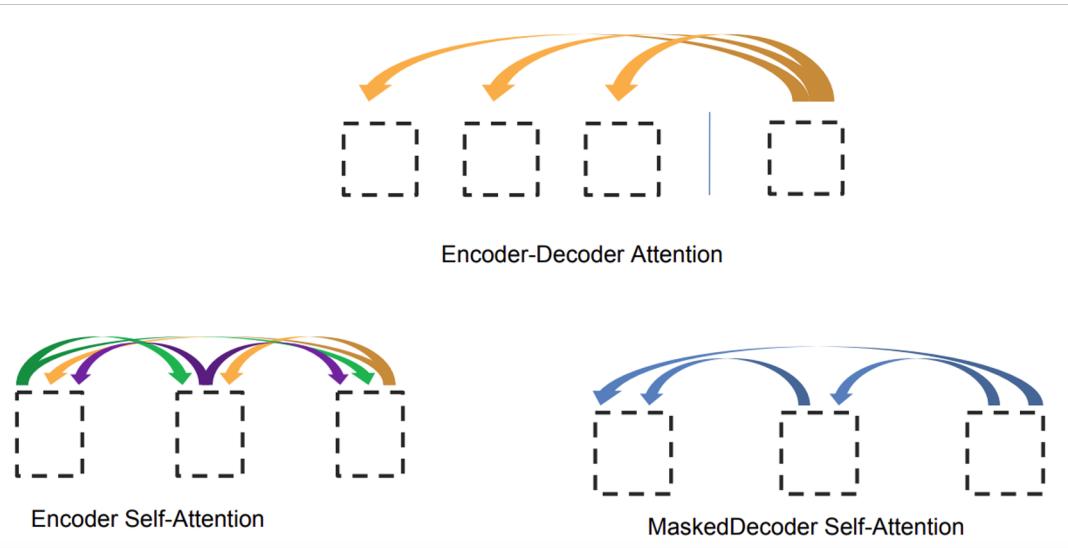
Attention is all you need =) 2017

Previously:

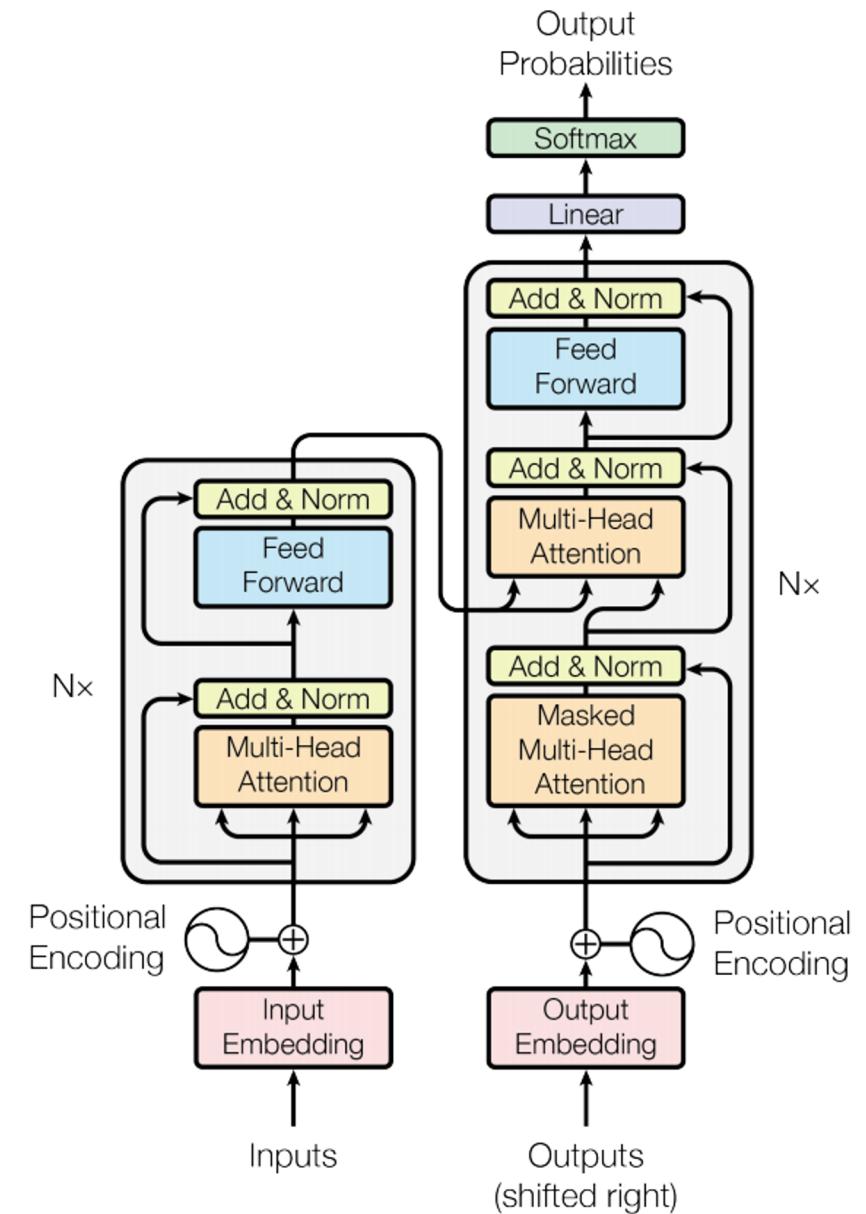
- RNN encoder + RNN decoder, interaction via fix-sized vector
- RNN encoder + RNN decoder, interaction via attention

NOW:

- **attention + attention+ attention**

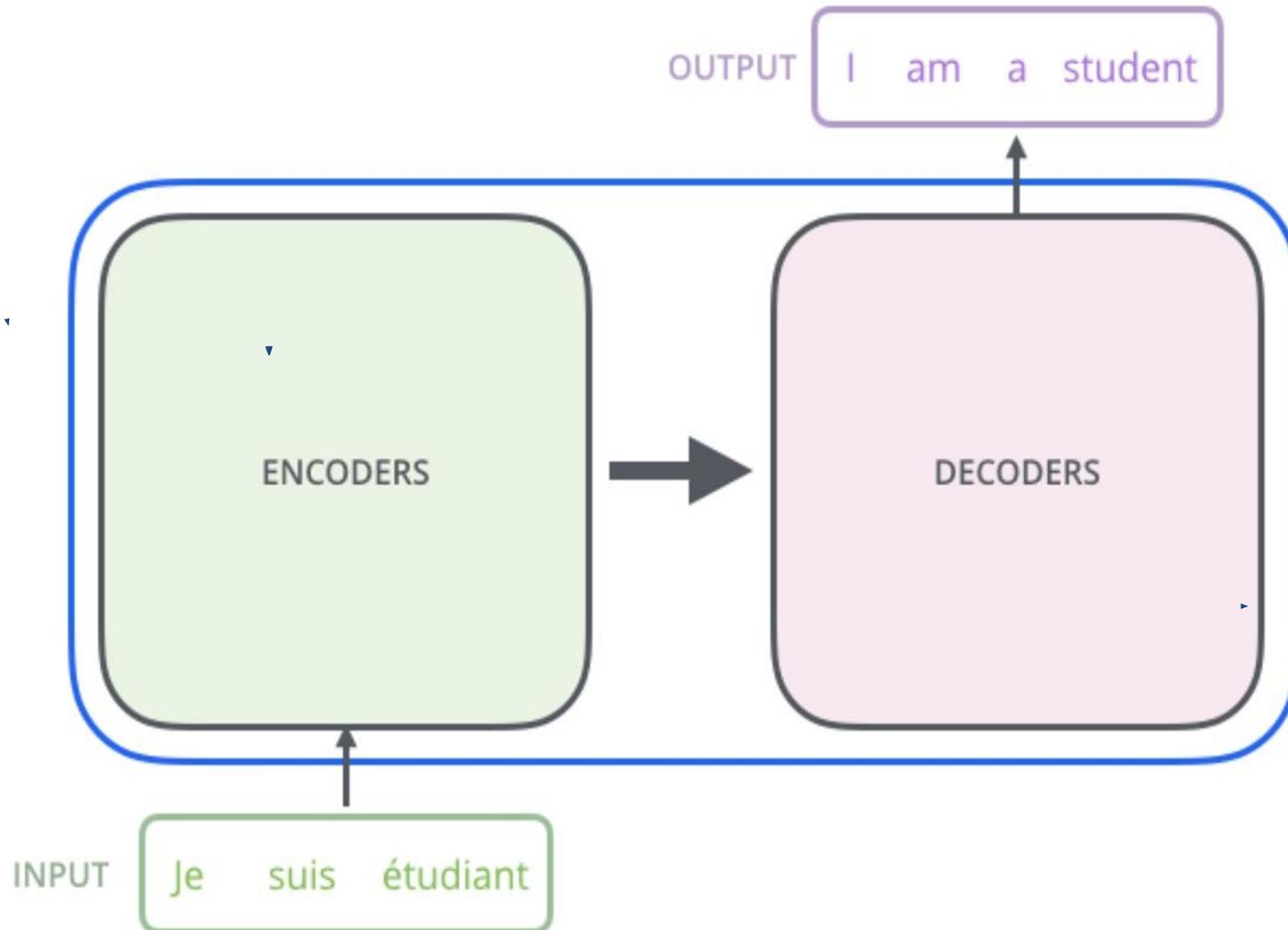


attention + attention+ attention

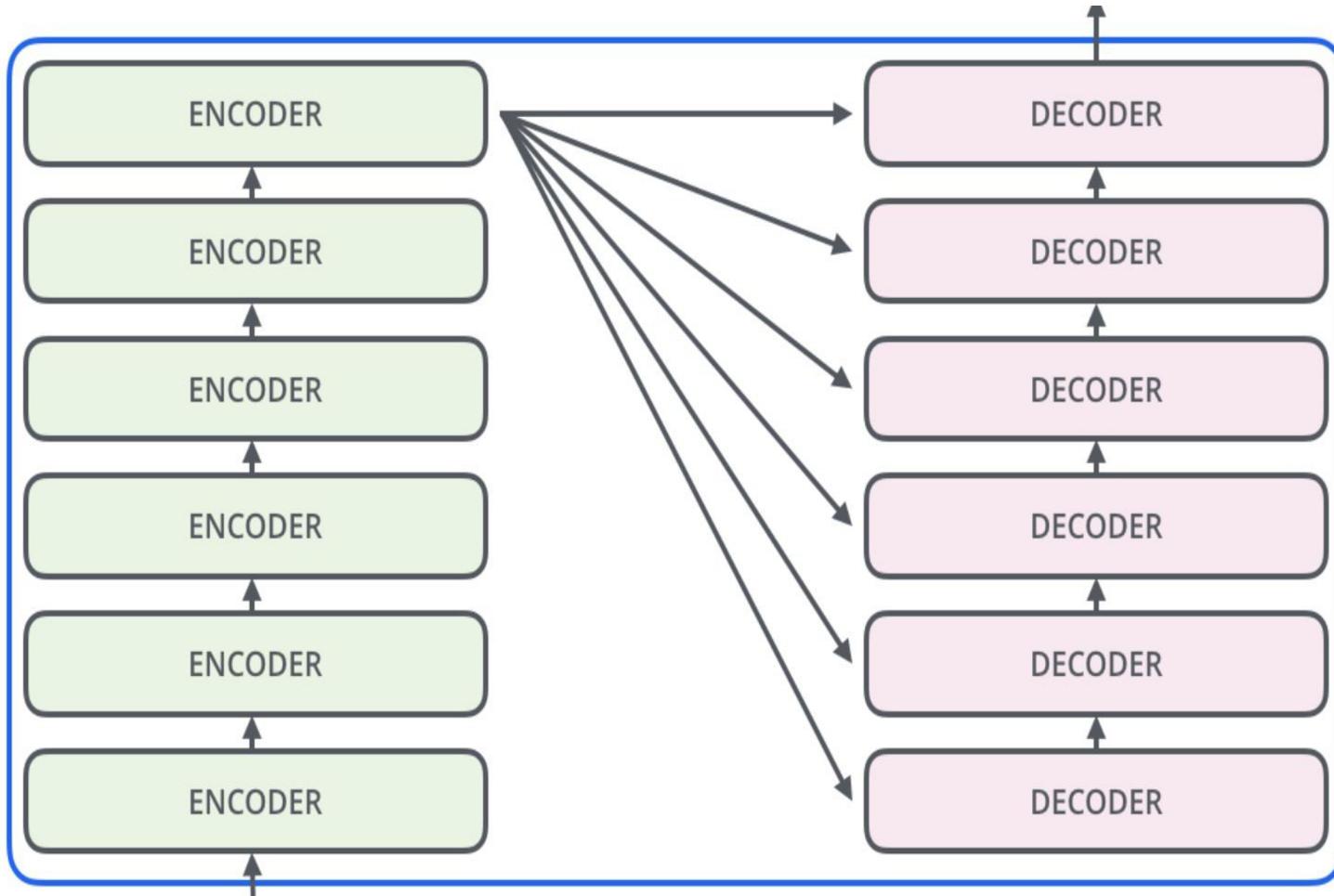


Lukasz Kaiser. 2017. Tensor2Tensor Transformers: New Deep Models for NLP. Lecture in Stanford University

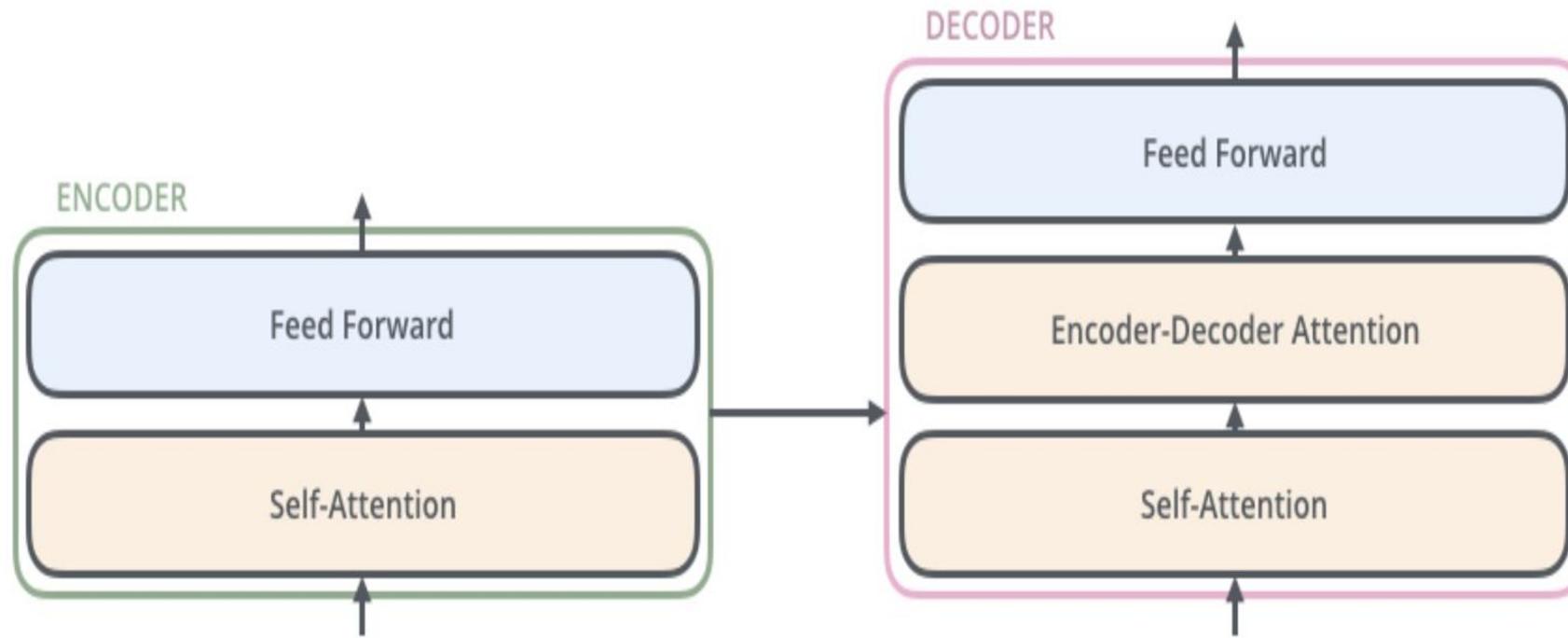
Transformer: high-level



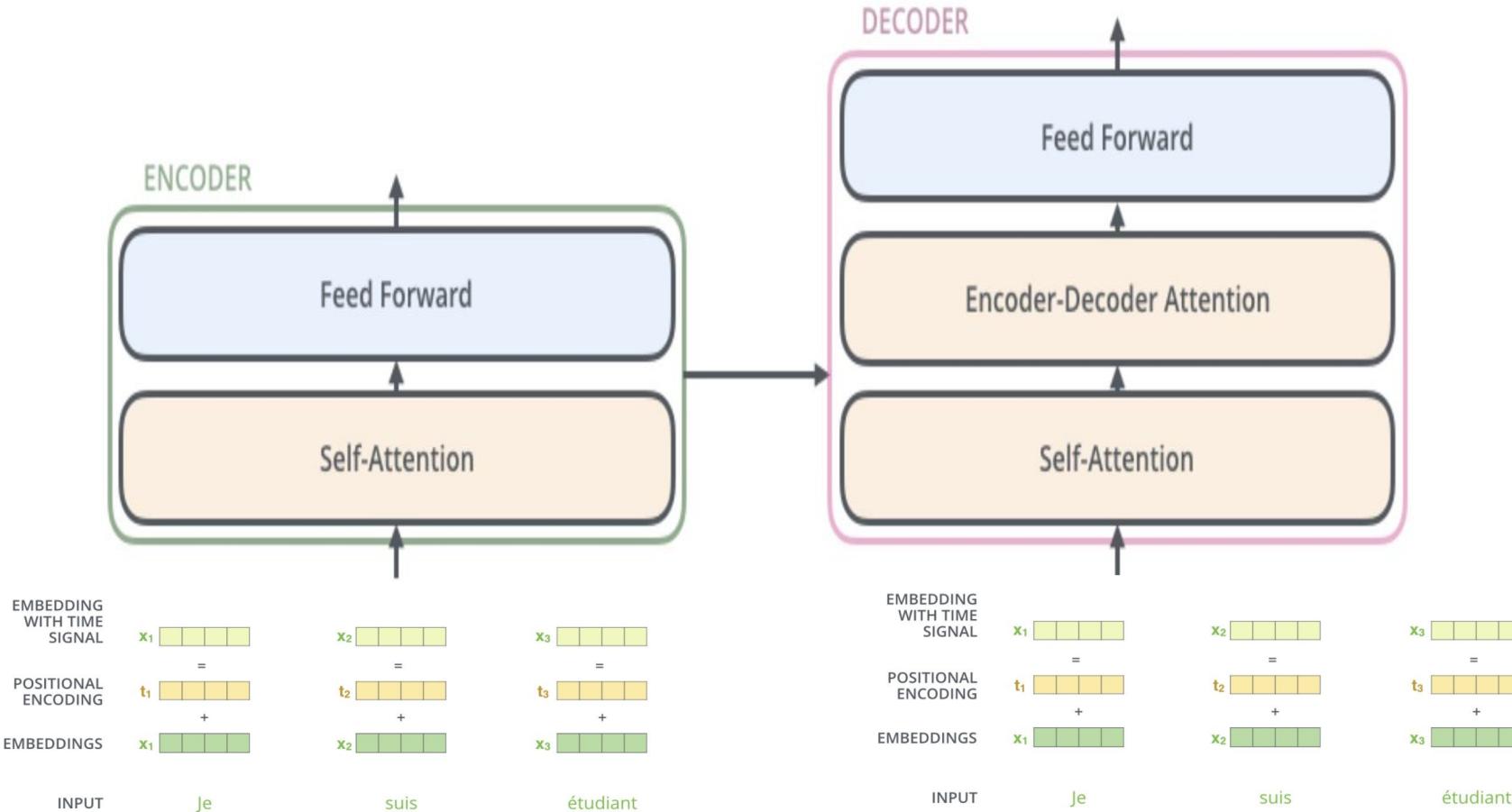
Transformer: high-level



Transformer: high-level

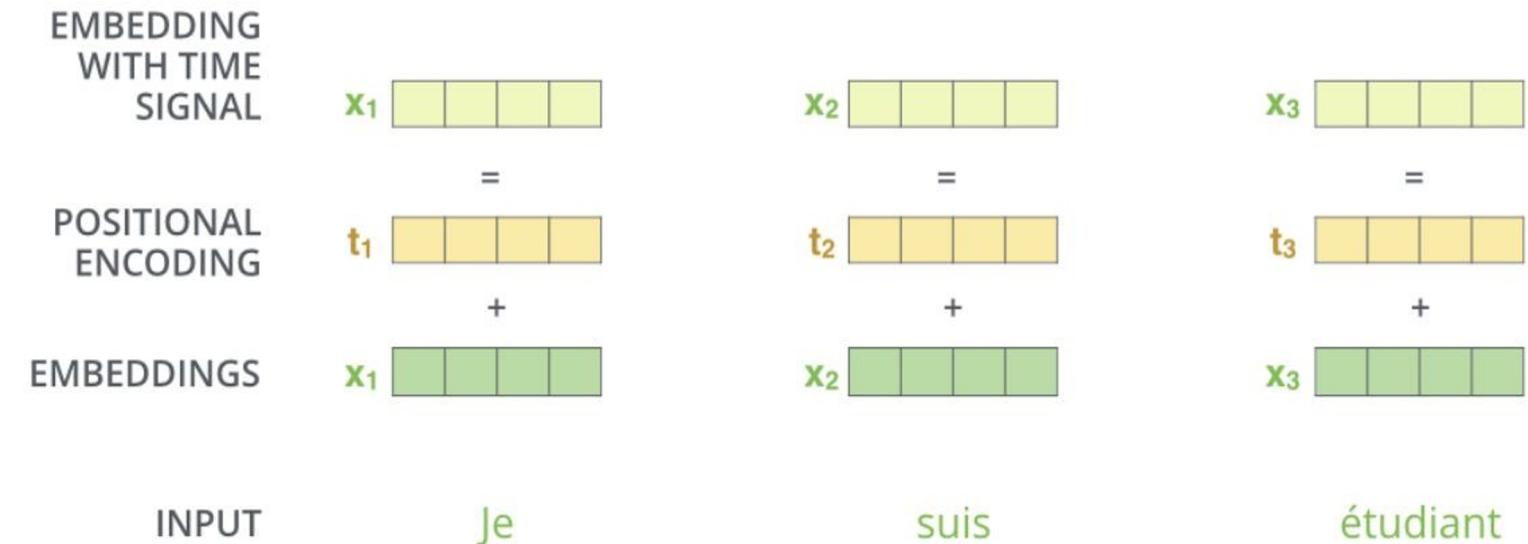


Transformer: high-level



Positional Encoding

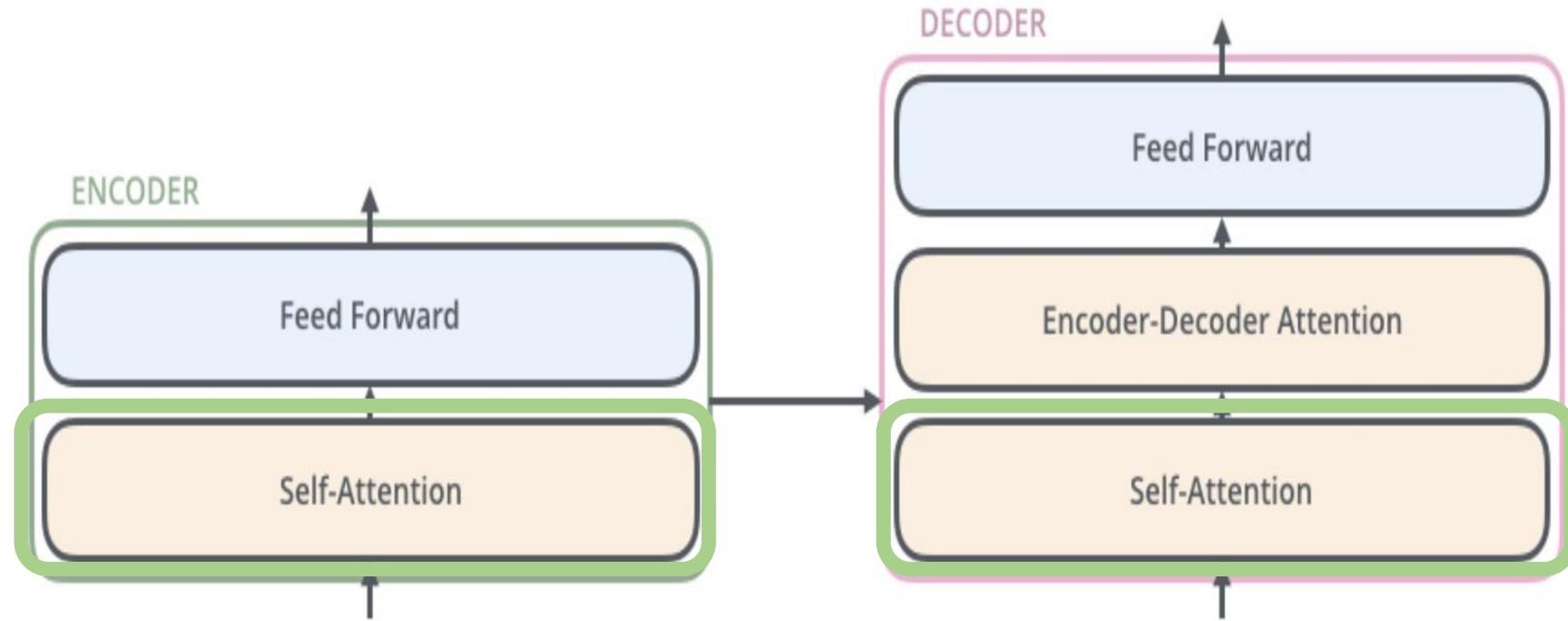
positional encoding provides *order information* to the model



The fixed positional encodings used in the Transformer

$$\text{PE}(i, \delta) = \begin{cases} \sin\left(\frac{i}{10000^{2\delta/d}}\right) & \text{if } \delta = 2\delta' \\ \cos\left(\frac{i}{10000^{2\delta/d}}\right) & \text{if } \delta = 2\delta' + 1 \end{cases}$$

Transformer: high-level



Transformers. Self-attention

Previously: one decoder state looked at all encoder

states **NOW:** each state looks at each other states

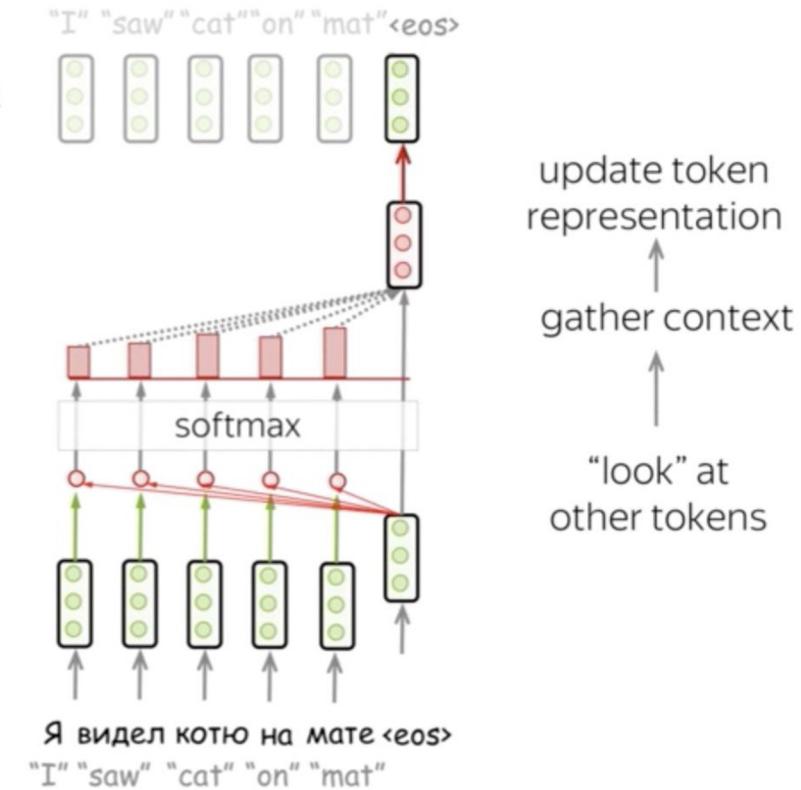
Self-attention:

- tokens interact with each other
- each token "looks" at other tokens
- gathers context
- updates the previous representation of "self"

Tokens try to understand themselves better in context of each other

Layer $k + 1$

Layer k



Query, Key and Value vectors

Query, Key and Value vectors

Each vector gets **three representations**:

- **query** - asking for information;
- **key** - saying that it has some information;
- **value** - giving the information.

These matrices allow different aspects of the x vectors to be used/emphasized in each of the three roles

Attention matches the key and query by assigning a value to the place the key is most likely to be.

$$\begin{matrix} \mathbf{x} \\ \times \\ \mathbf{W}^Q \\ = \\ \mathbf{Q} \end{matrix}$$

$$\begin{matrix} \mathbf{x} \\ \times \\ \mathbf{W}^K \\ = \\ \mathbf{K} \end{matrix}$$

$$\begin{matrix} \mathbf{x} \\ \times \\ \mathbf{W}^V \\ = \\ \mathbf{V} \end{matrix}$$

↓

Attention weights

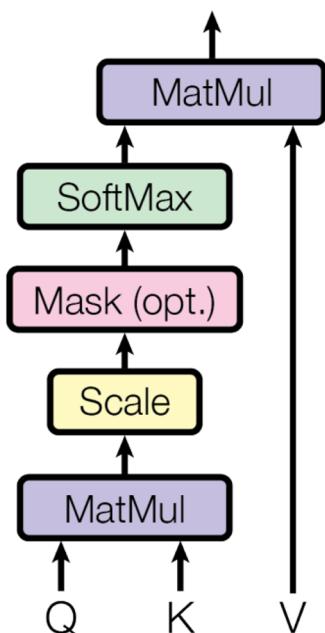
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Masked self-attention

Decoder has different self-attention => **Masked self-attention**

- we generate one token at a time => during generation, we don't know which tokens we'll generate in future.
- to enable parallelization we forbid the decoder to look ahead - future tokens are masked out (setting them to **-inf**) before the softmax step in the self-attention calculation

(Masked) scaled dot-product impl.



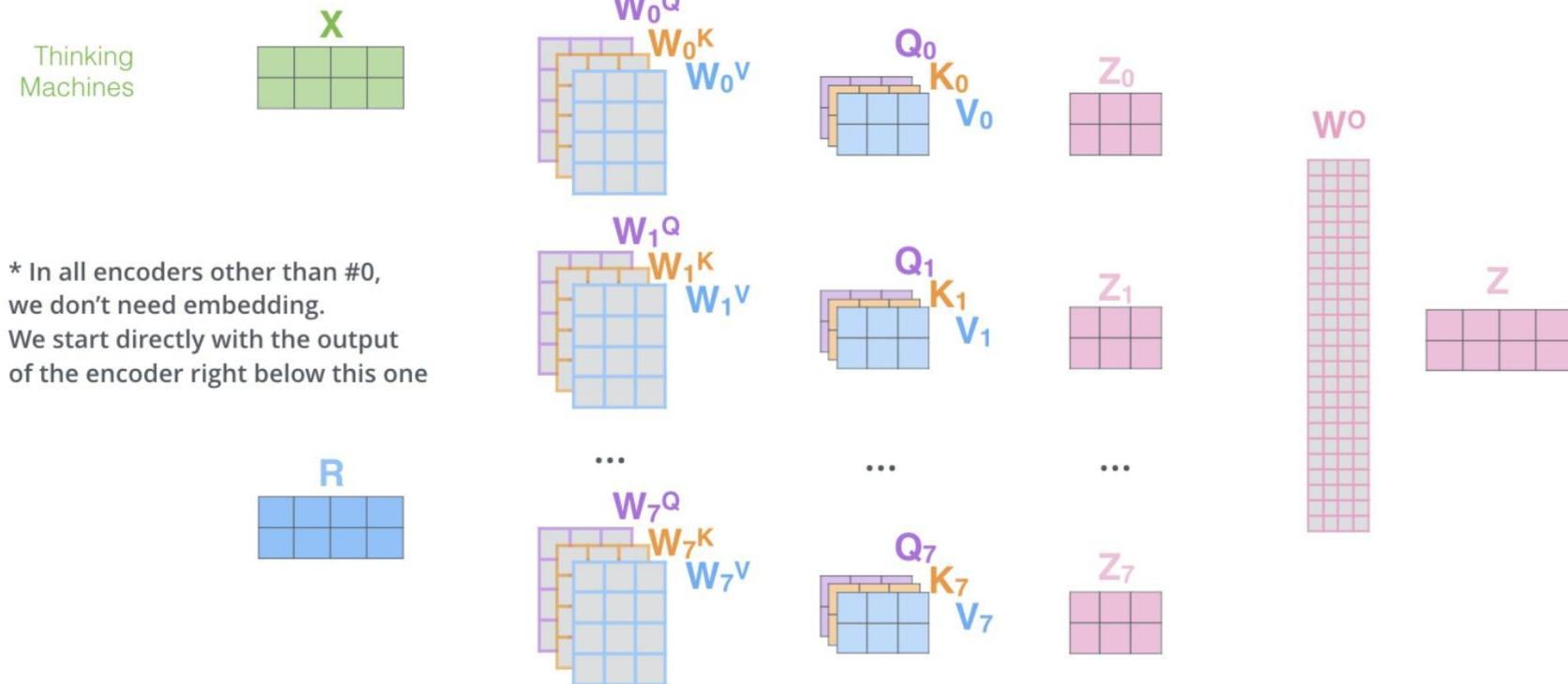
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    key_t = key.transpose(-2, -1)
    scores = torch.matmul(query, key_t) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Multi-head attention

- We need to know relationships between in a sentence: relationships, preferences, order, issues like case or agreement
- Instead of having one attention **multi-head attention** several "heads" which **independently** and focus different things.

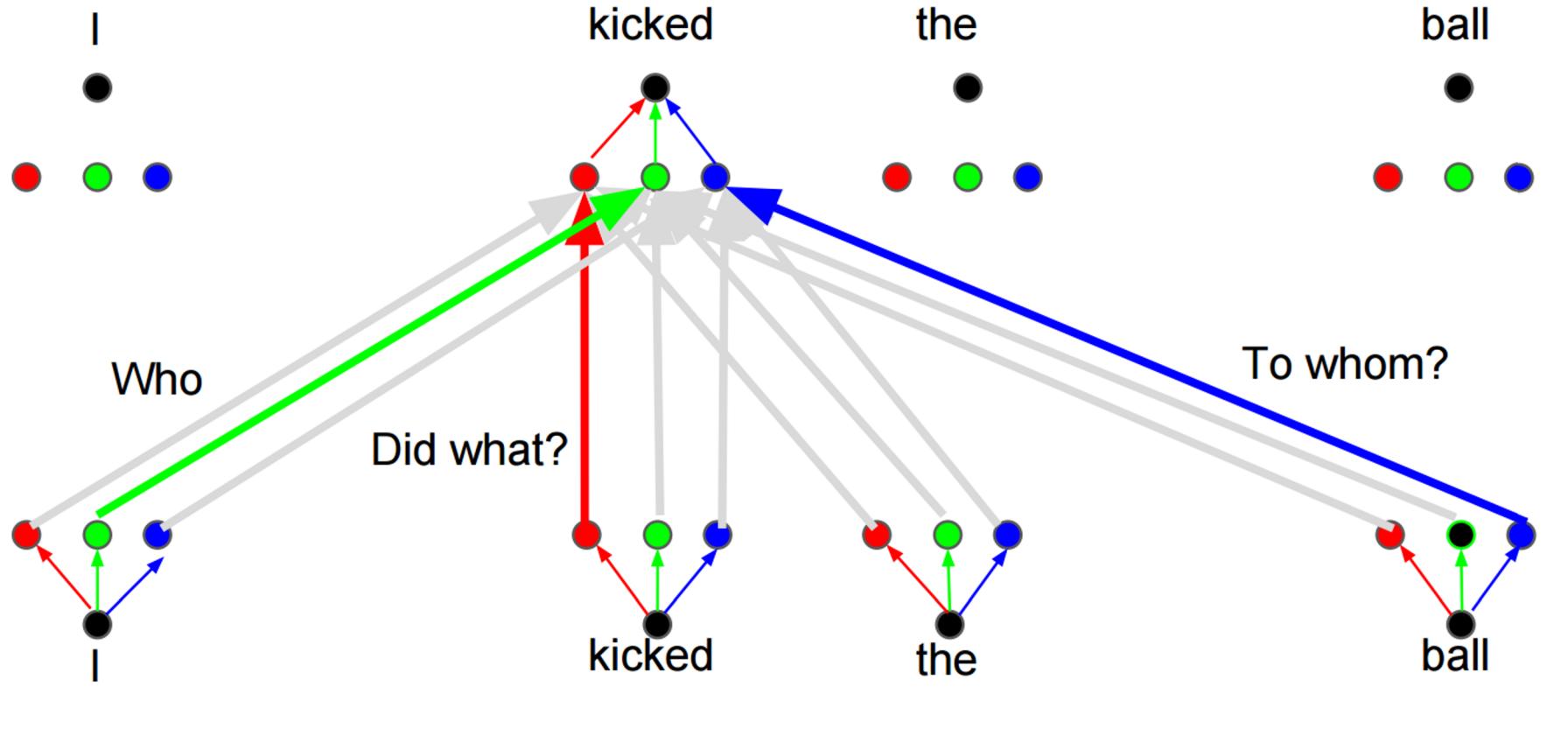
- 1) This is our input sentence* each word*
- 2) We embed
- 3) Split into 8 heads. We multiply \mathbf{X} or \mathbf{R} with weight matrices
- 4) Calculate attention using the resulting $\mathbf{Q}/\mathbf{K}/\mathbf{V}$ matrices
- 5) Concatenate the resulting \mathbf{Z} matrices, then multiply with weight matrix \mathbf{W}^O to produce the output of the layer



$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)\mathbf{W}^O$$

$$\text{where } \text{head}_i = \text{Attention}\left(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\right)$$

Multihead attention



Multihead attention impl

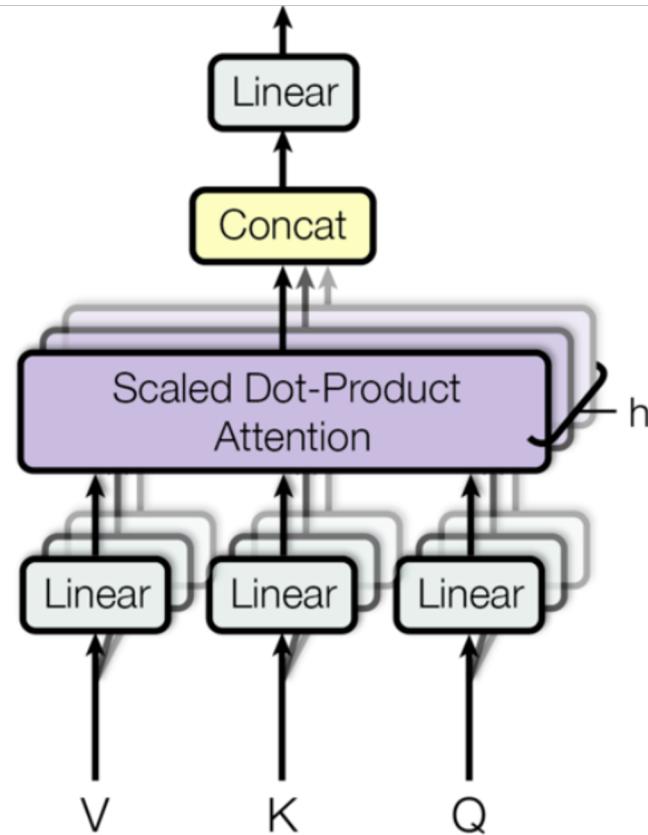
```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

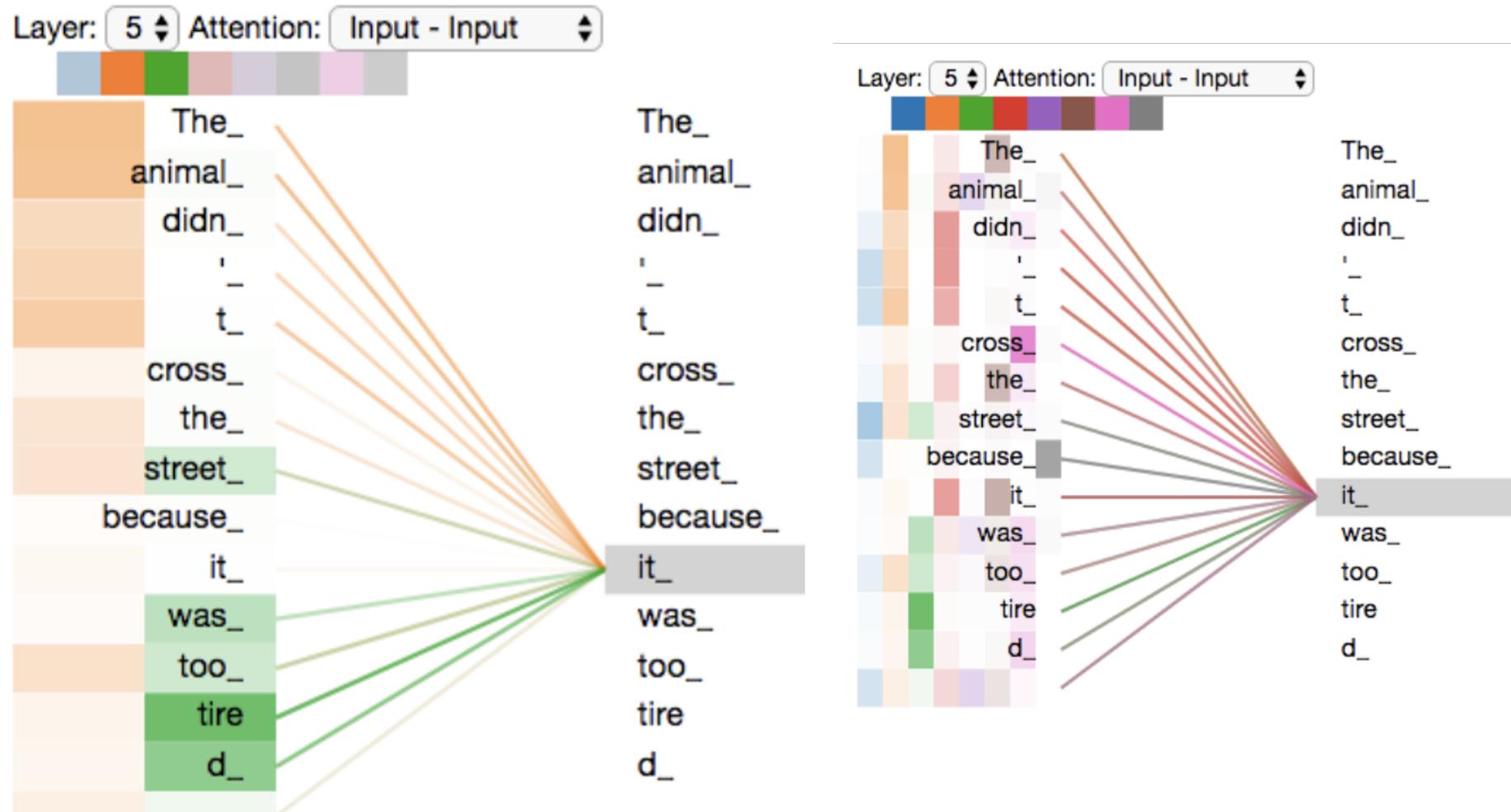
        # 1) Do all the linear projections in batch from d_model => h x
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)
        →(bs, 8, m, 64), (bs, 8, m, n)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
            .view(nbatches, -1, self.h * self.d_k) →(bs, m, 512)
        Concat: View as (bs, m, 512)
return self.linears[-1](x) Result is (bs, m, 512)
```



Multihead self-attention in encoder



Extra

Feed-forward blocks

each layer has a feed-forward network block: two linear layers with ReLU non-linearity between them

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2.$$

Positionwise FFNN

- Linear→ReLU→Linear
 - Base: 512→2048→512
 - Large: 1024→4096→1024
- Equal to 2 conv layers with kernel size 1

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

```
class PositionwiseFeedForward(nn.Module):  
    "Implements FFN equation."  
    def __init__(self, d_model, d_ff, dropout=0.1):  
        super(PositionwiseFeedForward, self).__init__()  
        self.w_1 = nn.Linear(d_model, d_ff)  
        self.w_2 = nn.Linear(d_ff, d_model)  
        self.dropout = nn.Dropout(dropout)  
  
    def forward(self, x):  
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

Extra

Feed-forward blocks

each layer has a feed-forward network block: two linear layers with ReLU non-linearity between them

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2.$$

Residual connection (train better)

Residual connections => add an input of the block to its output

Ease the gradient flow through a network and allow stacking a lot of layers

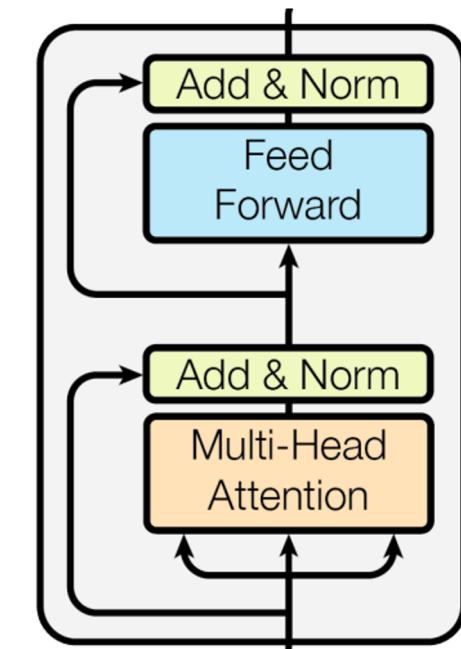
Residuals

$$\text{LayerNorm}(x + \text{dropout}(\text{Sublayer}(x)))$$

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```



Extra

Feed-forward blocks

each layer has a feed-forward network block: two linear layers with ReLU non-linearity between them

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2.$$

Residual connection (train better)

Residual connections => add an input of the block to its output

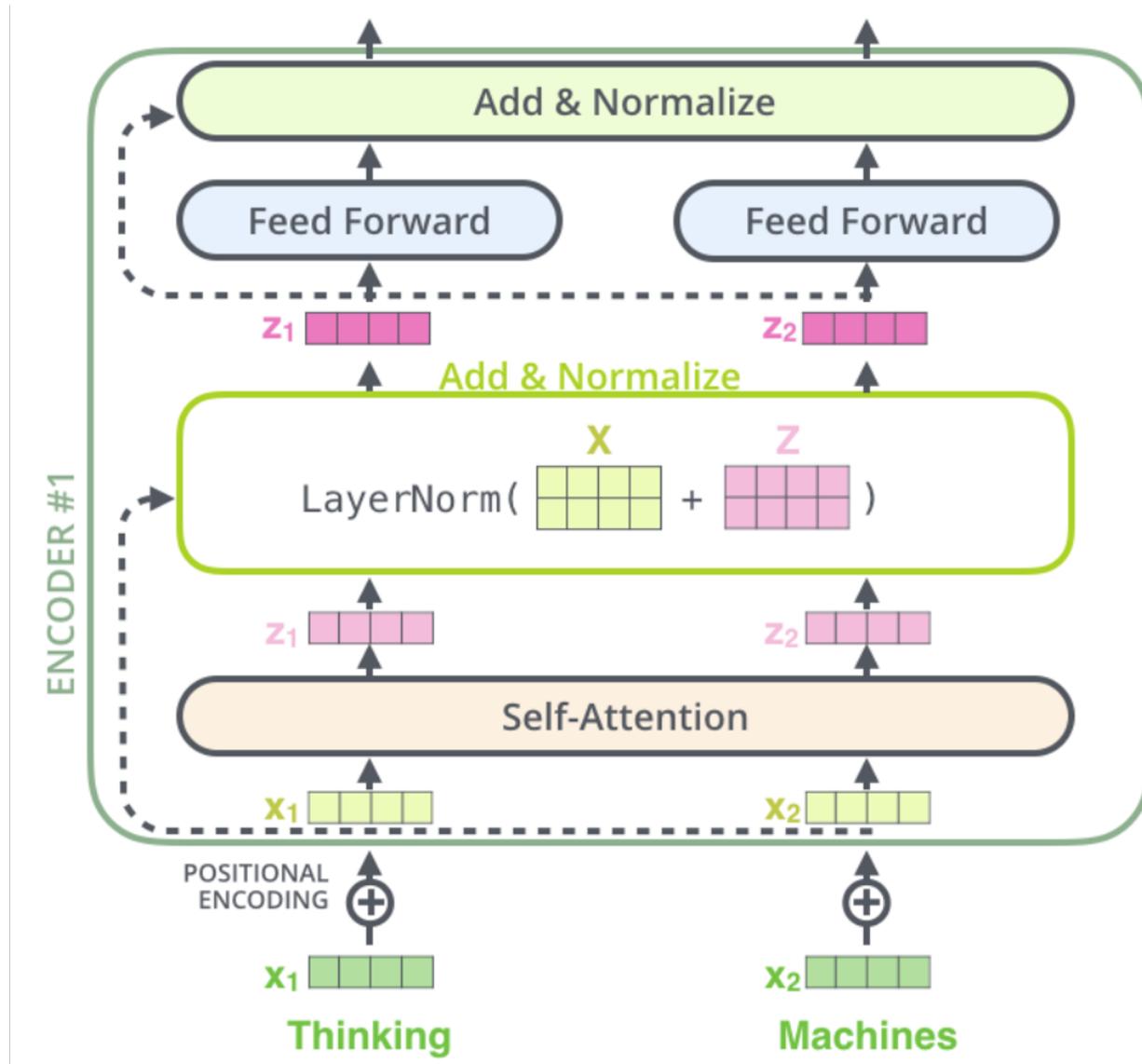
Ease the gradient flow through a network and allow stacking a lot of layers

Layer Normalization (train faster)

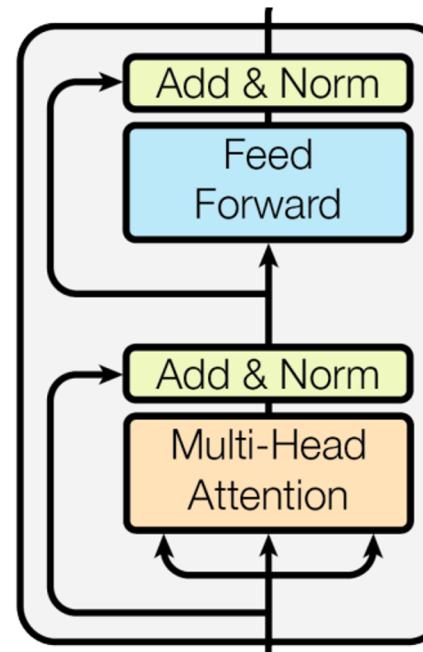
Improves convergence

Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation within each layer

Transformer layer (enc) unrolled



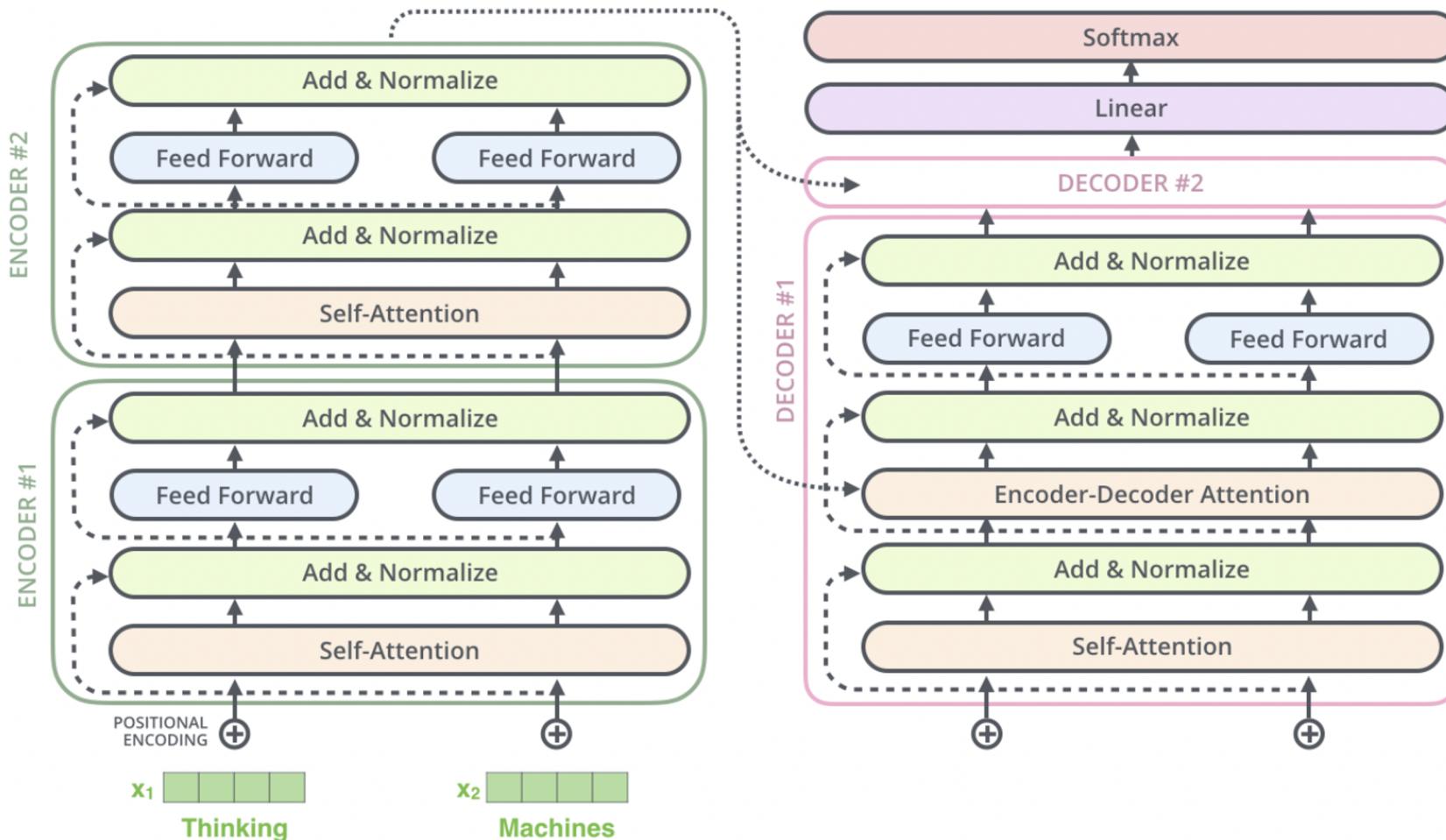
Transformer layer (enc)

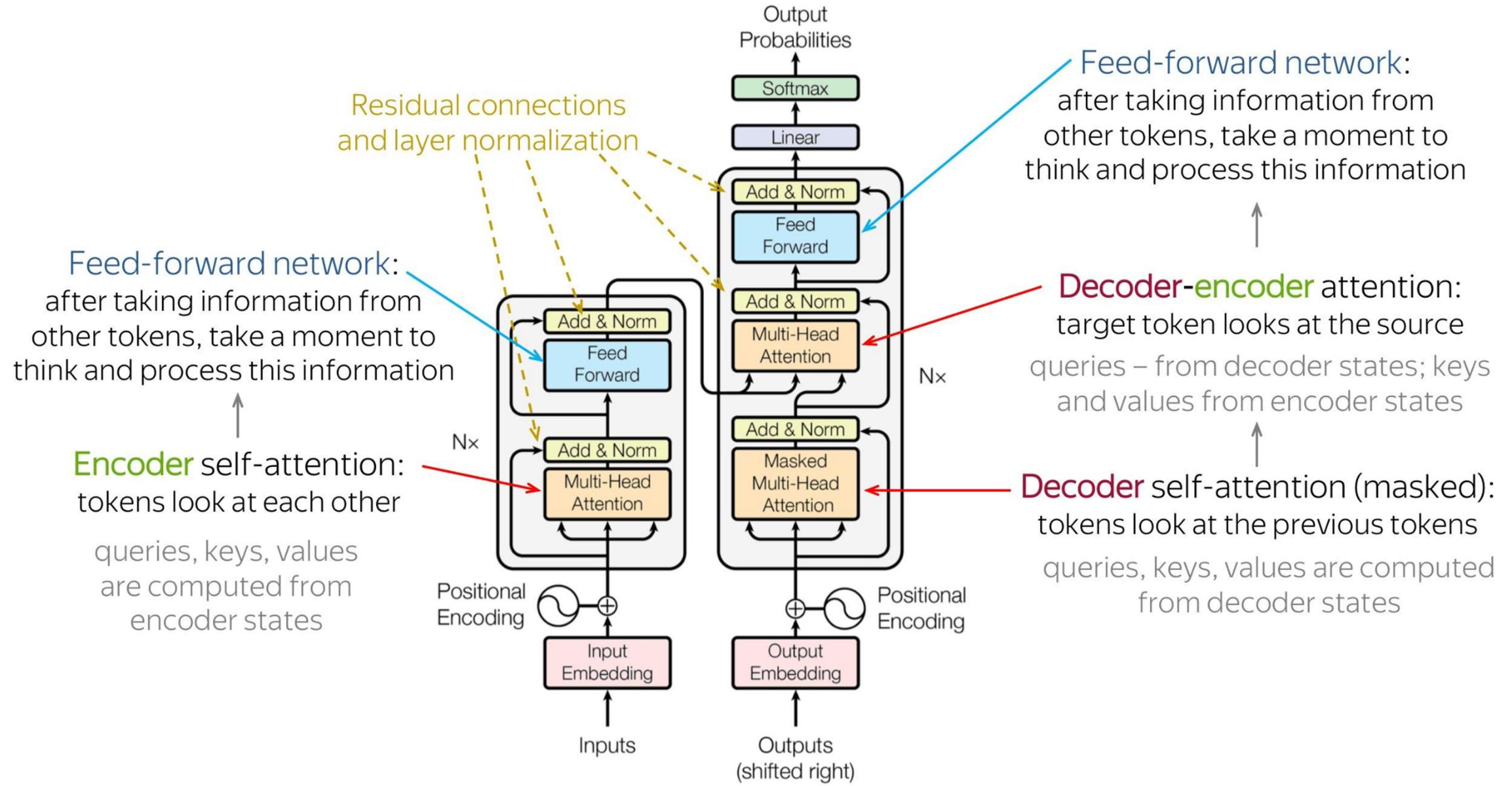


```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Transformer layer (dec) unrolled





Regularization

- **Dropout:**
 - *residual*
 - *ReLU*
 - *Input*
- **Attention dropout (only for some experiments)**
 - *Dropout on attention weights (after softmax)*
- **Label smoothing**

Label smoothing from:

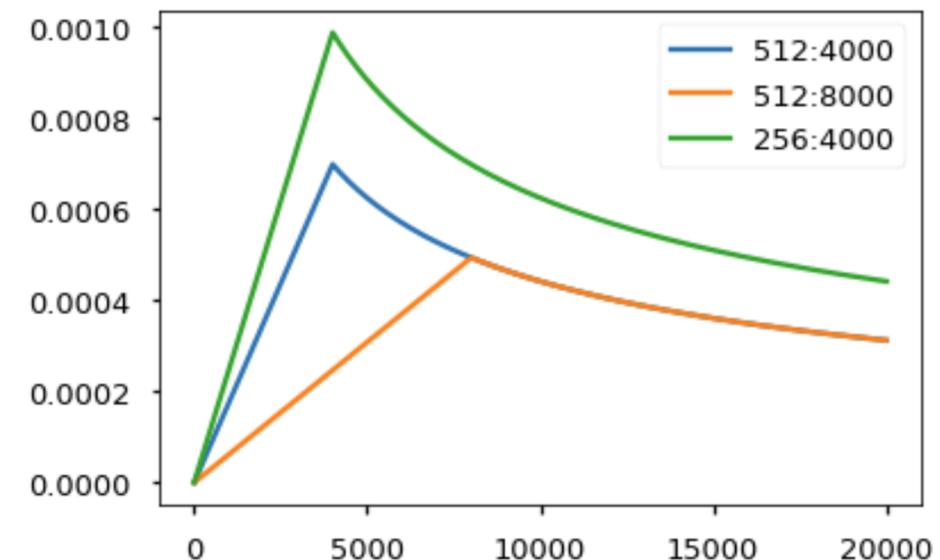
Szegedy. Rethinking the Inception Architecture for Computer Vision, 2015

Training

- Adam, betas=0.9,0.98, eps=1e-9
- Learning rate: linear warmup: 4K-8K steps (3-10% is common) + square root decay

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step_num^{-0.5}, step_num \cdot warmup_steps^{-1.5})$$

**Noam Optimizer:
Adam+this lr schedule**



Training

- WMT2014 En→De / Fr: 4.5M / 36M sent.pairs
 - *word-pieces vocab: 37K shared / 32K x2 separate*
 - *Batches: sequences of approx. same length, dynamic batch size: 25K src & 25K tgt tokens*
 - *On 8 P100 GPU (16GB), base/big: 0.5/3.5 days, 100k/300k steps 0.4/1.0s per step*
 - *Average weights from last 5/20 checkpoints*
 - *Beam search with size 4, length penalty 0.6*

Dev: newstest2013 en→de

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
big	6	1024	4096	16			0.3		300K	4.33	26.4	213

Results

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.8	$2.3 \cdot 10^{19}$	

Transformers 6 years later

What happened after “Attention is all you need”?

- Mostly used for language modeling
- It turned out than pretrained transformers are easy to **fine-tune for a new task**
 - Self-attention seems to learn a lot about the structure of language
 - The architecture has enough capacity to generalize to new tasks
- A lot of **pre-trained transformers** have appeared
 - They are now among the most important NLP resources
 - Many of them are public (e.g. <https://huggingface.co/models>)
- For many tasks, it is easier (and often more effective) to **fine-tune an existing transformer** than to devise and train a model from scratch

Language Modelling

- **Language modeling** is the task of predicting what word comes next
 - *The student opened their ____ (books | laptops | exams | minds |...)*
 - *Что дальше будет неизвестно ____ (никому | заранее | ...)*

Language Modelling

- **Language modeling** is the task of predicting what word comes next
 - *The student opened their ___ (books | laptops | exams | minds | ...)*
 - *Что дальше будет неизвестно ___ (никому | заранее | ...)*
- **Formally:** given a sequence of words x_1, x_2, \dots, x_t , compute the probability distribution of the next word x_{t+1}
- A system that does it is called a **language model (LM)**
 - A probabilistic multi-class classifier: $P(x_{t+1} | x_1, x_2, \dots, x_t)$

Language Modelling

- : **Language modeling** is the task of predicting what word comes next
 - *The student opened their ___ (books | laptops | exams | minds | ...)*
 - *Что дальше будет неизвестно ___ (никому | заранее | ...)*
- **Formally:** given a sequence of words x_1, x_2, \dots, x_t , compute the probability distribution of the next word x_{t+1}
- A system that does it is called a **language model (LM)**
 - A probabilistic multi-class classifier: $P(x_{t+1} | x_1, x_2, \dots, x_t)$
- **Usages:**
 - Reranking hypotheses of a translation model (e.g. in IBM models)
 - Generating a translation (if the model is conditional on the input text)

n -gram Language Modeling

- First we make a **simplifying assumption**: $\mathbf{x}^{(t+1)}$ depends only on the preceding $n-1$ words.

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)}) = P(\mathbf{x}^{(t+1)} | \underbrace{\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}}_{n-1 \text{ words}}) \quad (\text{assumption})$$

$$\begin{aligned} \text{prob of a n-gram} &\rightarrow P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}) \\ \text{prob of a (n-1)-gram} &\rightarrow P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}) \end{aligned} \quad \begin{aligned} & \quad (\text{definition of conditional prob}) \end{aligned}$$

- Question:** How do we get these n -gram and $(n-1)$ -gram probabilities?
- Answer:** By **counting** them in some large corpus of text!

$$\approx \frac{\text{count}(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}{\text{count}(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})} \quad (\text{statistical approximation})$$

Disadvantages of n-gram LMs

- needs smoothing
- can't handle long history
- they cannot generalize over contexts of similar words well

Neural Language Models: Motivation

- (+) a neural language model has much **higher predictive accuracy** than an n -gram language model!
- (-) neural net language models are **strikingly slower to train** than traditional language models

Basic types of transformer architectures

BERT-like models: encoder only

			мы	##ла						
...
...
[CLS]	M	##ама	[MASK]	[MASK]	p	##ам	##y	.	[SEP]	

GPT-like models: decoder only

M	##ама	мы	##ла	p	##ам	##y	.	<e>
...
...
<s>	M	##ама	мы	##ла	p	##ам	##y	.

The classical transformer (also T5, BART, etc.): encoder and decoder

...
...
<s>	M	##ама	мы	##ла	p	##ам	##y	.	<e>	



Mo	##m	was	was	##hing	the	frame	Mo	<e>
...
...
<s>	Mo	##m	was	was	##hing	the	frame	.

In encoders, each token attends **to each other token**

They are good for understanding texts.

In decoders, each token attends **only to the tokens on the left** (and to all tokens of the encoder, if it exists)

They are good for generating texts.

Do you really want a transformer?

- **Probably yes, if:**
 - There is a pretrained transformer for your language and task (or a similar one)
 - You have a small training set and want to generalize from it
 - Your task is natural language generation and you have no templates
- **Probably no, if:**
 - Your problem can be solved with rules or keywords
 - You need fast (around 1ms) inference without GPU
 - You need fast training without GPU
 - You have to process very long texts without splitting them
 - Your task needs some specific architecture (e.g. memory or recursion)